Intel® Protected Access Architecture

Application Interface Specification, Revision 1.0

March 2001

Revision History

Revision	Revision History	Date
0.5.1	Initial Revision.	07/21/1999
0.7.5	Revisions include: • Added Protected Storage Interface.	01/07/2000
0.9.0	Revisions include: • Storage and Token interfaces reworked based on review feedback. Added storage initialization functions, added device level of operation. • Added Device configuration Interface • Changed definition of function descriptor and use with InitializeFN • Added description of protected storage to introduction • Changed name from PAS to Intel® Protected Access Architecture	02/10/2000
0.9.1	Revisions include: • Fixed contact information and added feedback deadline to front matter	03/14/2000
0.9.5	Revisions include: • Added storage Lock/Unlock functions • Updated Permission/Access control object specification and description; • Changed tkConnectFn to tkLoginExtFn to be more explicit and communicate it relationship to tkLoginFn better.	09/01/2000
1.0RC1	Revisions include: • Changed storage Lock/Unlock functions • Changed storage Set Access function • Added Access Rights for storage loac / unlock operations • Updated Token Initialize function – concept of subdivision data on sections enchanced and clarified.	12/19/2000
1.0RC2	Revisions include: • Changed definition of storage Allocate and Free functions – both have been made priviledged to avoid breach of security when low priviledged caller allocates and uses slot with high priviledges.	12/26/2000
1.0RC3	Revisions include: • Changed storage lock/unlock masks – now masks are better correlated with each other. • Revised and corrected error codes. • Updated set of common tags. • Added ACO update protection mechanism • Added message class	4/27/2001
1.0	Revisions include: • Register function added to Token SPL and new function set Register-Capture-Match is created to cover scenarios when user template capturing at enrollment time differs from one at verification time.	5/8/2001

Contents

Figures

Tables

1.0 Abstract

This document outlines a method to help reduce PC theft by “strengthening” user authentication during the PC “boot” process. It describes an architecture, a high-level programming interface to authentication devices and protected storage, as well as common interface elements needed to support the high level interface. It is intended that PC BIOS programmers and authentication device manufacturers can use this specification to help make pre-boot user authentication solutions compatible across PC platforms.

2.0 Introduction

Intel recognizes the need to work with the industry to develop security standards and identify, develop, and implement building blocks to enable them. One trend in progress in the industry is the increased reliance on mobile computers to provide workers with a PC platform that can be used at work, on the road, or at home. However, mobile PC platforms have a unique security issue in that they are frequently exposed to theft, both for the value of the platform itself and for the data it contains. In fact, studies done by both Safeware*(97) and Chubb*(98) indicate that the value of stolen mobile PCs has reached $1B/year and that the data they contain is estimated at $15B (Chubb 98). To find out which security issues were deemed the most important, Intel conducted two Internet focus group studies (IT55 9/98, F1000 9/97). This data confirmed that mobile PC security was a top concern for IT managers. Specifically, it showed that among the top mobile concerns was asset and data theft. Lastly, respondents indicated that a single solution should be applicable across all computer platforms, not just mobile PCs. The solution presented in this specification, though targeted at mobile platforms, is intended to have utility for desktop and server platforms as well.

There are some devices on the market today that are designed specifically to help prevent mobile PC theft. These devices include:

- Cables used to lock the platform to something solid
- Motion detectors or “radio leashes” that scream when they detect tampering or get too far from their owner
- “Phone Home” software that checks in to “Home Base” on a regular basis to report its location
- Software that modifies the boot block of the hard disk to require the user to “login” before allowing OS-load to continue
- OS-present software that is designed to authenticate the user with stronger means than usual. All of these have their strengths and weaknesses of course. But one will notice that all of them require something to be *added* to the system to enable the additional level of protection. What if that “something” was built-in and simply denied use of the platform to all unauthorized users. This would be a very effective means to deter theft. This approach is called “Pre-OS User Authentication”. Requiring multiple factors of identity rather than just one can strengthen Pre-OS User Authentication, such as requiring the user to provide two or more of the following:
  - Something you know (like a password)
  - Something you have (like a USB Token, Smart card, etc.)
  - Something you are (like a biometric measurement such as a fingerprint).

Since every PC platform has a system BIOS that contains code that is always executed first when power is applied or the system wakes up from certain sleep states, and since the BIOS code is usually stored in FLASH that is much less accessible than normal storage on the PC platform, the system BIOS is an ideal place to include code that comprehends the notion of and enforces “Strengthened User Authentication”.

To date, this has been accomplished with a simple BIOS password (something you know). Only recently has this method come under question as an adequate protection mechanism. This is because it is fairly easily circumvented either with some form of “social engineering”, guessing, accidentally exposing the password on a “sticky note”, or by a global secret that has been shared on the Internet.

Additionally, some forms of single factor authentication can improve system confidence in the user's identity and make the platform easier to use. One such case is the fingerprint, where one can't lose it and others can't steal it easily. In fact, once the pieces are put in place for fingerprint recognition on the PC platform, there is an opportunity to improve ease-of-use as well. This is because solutions can be provided where the user doesn't have to remember a multitude of passwords anymore.

Up until this point, we have been discussing mechanisms for improved asset theft deterrence, but what about data theft? Since the hard disk drive (HDD) can be easily removed from most personal computers (PCs) and since it is the typical long-term store for user's confidential and private data, a method of protection is needed here as well. There are basically two usable forms of data protection here. One is to prevent access to the data, like the lock on one's front door that prevents strangers from walking in. The other is to scramble the data so that it is useless to someone who doesn't know how to de-scramble it. Hard disk drives that have been manufactured to the ATA-3 standard (and above) usually have the first capability, known as hard disk password. This is basically a password that unlocks the drive so that it will respond to subsequent read/write commands. The second capability is rapidly becoming more available to users in the form of file and directory encryption/decryption software. This software has access to time tested algorithms that can provide the user with a high level of data protection. By using either or both of these approaches, the data is relatively useless to a casual thief.

A very important factor in supporting the above capabilities is the ability to store and protect secrets, such as encryption keys, on the PC platform. Thus there is a need for a new PC platform element, referred to in this spec as "Non-volatile Protected Storage". This new storage has various forms of access control and implementation, but it is designed to limit which programs and/or users have access to it.

Lastly, it has been shown in the past, especially in the media, that no matter how good a security solution is, it can usually be circumvented with enough knowledge, time, and money. The goal of a good security system then becomes providing enough security to make it too expensive for the average attacker to deal with. The goal of this specification is to define a software interface , and its supporting characteristics, that allow implementations to meet this objective.

2.1 Glossary and Acronyms

Table 1: Glossary of Terms

Term	Definition
API	Application Programming Interface
APL	Application Programming Library
AuD	Authentication Device
Authentication Data	Data generated by an authentication service provider that will be used later to verify the identity of the user. Can be fingerprint templates, certificates, passwords, etc.
BDK	BIOS Developer's Kit
BIOS	Basic Input/Output System
BIS	Boot Integrity Services
DCI	Device Communication Interface
HDD	Hard Disk Drive
ICV	Integrity Check Value – A digital value obtained by hashing data. This value can be used to verify the integrity of the data later. This is it can detect if it has been altered in any way.
POST	Power-on Self Test. The Pre-boot environment between power on and the booting of the operating system. The BIOS controls the system during POST.
Pre-boot Application	A 3rd party application such as an option ROM or the BIS, which runs prior to booting the operating system. Applications, which may run during an ACPI S3 and/or S4 resume, are also referred to as Pre-boot applications in this document.
PXE	Pre-boot Execution Environment
Security System	A System that contains IPAA services and conforms to this specification.
SPI	Service Provider Interface
SPL	Service Provider Library
Storage SPL	The 3rd party SPL which responds to the IPAA API and communicates to the storage medium.
Template	Used loosely in this document to refer to data returned by the authentication service provider that will later be used to verify the identity of a user.
Token SPL	The 3rd party SPL which responds to the IPAA API, performs authentication operations, and communicates to the authentication device hardware.
WfM	Wired for Management
IBV	Independent BIOS Vendor
ISV	Independent Software Vendor
IHV	Independent Hardware Vendor
IPAA	Intel® Protected Access Architecture

2.2 Intended Audience

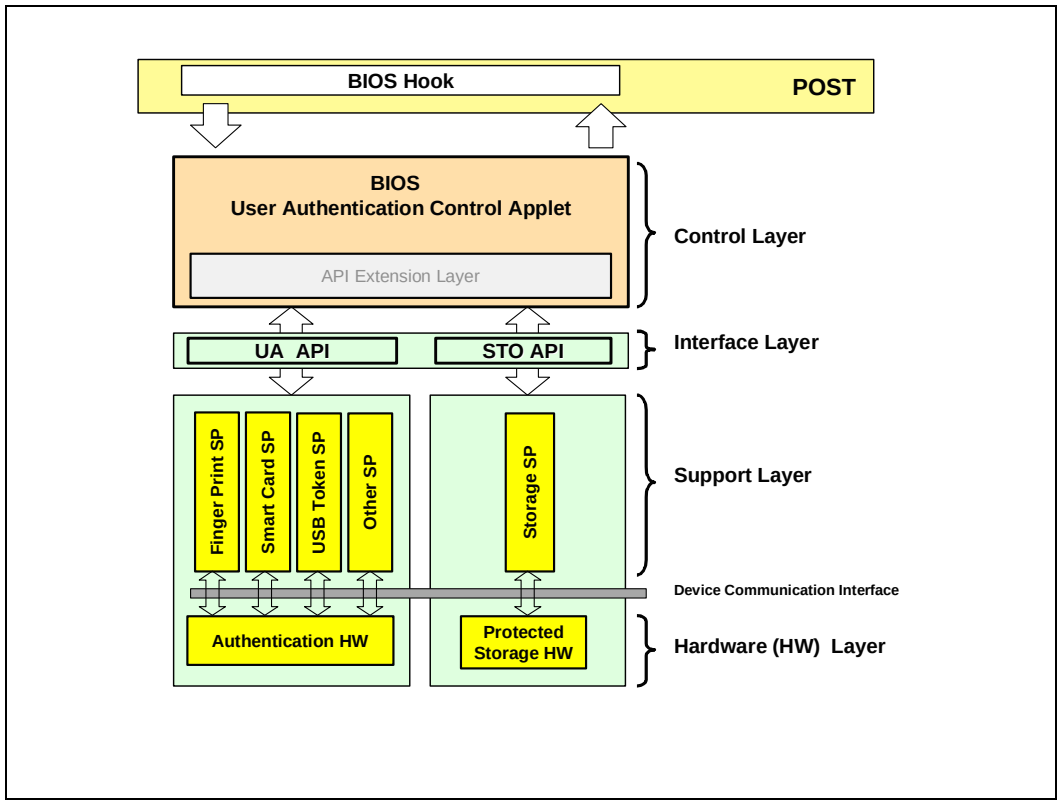
This specification is intended for use as a reference to the proposed IPAA authentication architecture and interface by technical implementers and engineering managers from Independent Software Vendors (ISVs), Independent Hardware Vendors (IHVs) that make authentication devices, Independent BIOS Vendors (IBVs), and OEMs who write their own BIOS.

3.0 Architecture

3.1 Platform Architecture

The high-level architecture for strengthening pre-boot user authentication is shown below in Figure 1.

Figure 1: Pre-boot User Authentication Architecture



The top line represents the BOOT process from the time the power is turned on (or during certain resume sequences) until control is passed to the OS. At some point during the power on self-test (POST) execution, the BIOS can be configured to authenticate the user attempting access. As mentioned before, this has normally been done by requesting a password from the user. In this architecture the task of user authentication is instead, passed to a new "user authentication stack" which consists of the following 4 key layers:

- Control Layer
- Interface (API) layer
- Support Layer
- Hardware Layer

Currently the User Authentication APIs are proprietary to individual IHVs, IBVs, or OEMs. This architecture proposes a solution that simplifies implementation and deployment while offering opportunity for third party differentiation. The goal is that the defined user authentication APIs will eventually be accepted as a minimal subset of an industry standard.

3.1.1 Control Layer (Pre-OS Authentication Control Applet)

This layer is the master control for pre-boot user authentication. It is owned and written by the BIOS writers of the system (BIOS vendor or OEM). It has several responsibilities that include:

- Finding, Interpreting, and Enforcing Platform Security Policy
- Finding and Configuring Platform Authentication and Storage Sub-systems
- Managing Authentication Modules and Entry Points
- Standards Based Memory Management for Pre-OS Authentication
- API Parameter Checking and Conversion
- Traffic Control Between Authentication and Storage Sub-systems
- Other BIOS Specific Details
- Optional BIOS Specific IPAA API Extensions
- Support Enabling/Configuring necessary resources for authentication

3.1.1.1 Platform Policy

Platform policy has three components. That which is enabled by the manufacturer, that which is selected by the platform owner, and that which is selected by the user of the system. The IPAA BIOS applet must find, interpret and enforce the policy. Policy can define how to handle specific security related events such as:

- No one currently registered as user
- Remote LAN wakeup
- Resume from suspend
- Privileges of current user (in BIOS space).
- Number of failed access attempts
- ATA-3 Locked
- Etc. (Serviceability, Manageability, Recoverability)

This policy would typically be stored in the system's non-volatile memory, preferably in the protected storage, and configured from the BIOS SETUP screen. Although, some implementations may have OS-present applications that provide easier control of these settings.

3.1.1.2 Find and Configure

The control applet needs to know which means of authenticating the user are present. Thus the control applet is responsible for enumerating the SW and HW capabilities of the platform. It can then compare the authentication mechanisms found with those needed by the platform policy and make decisions on how to proceed.

3.1.1.3 Module Management

Assuming the implementation on a PC platform is modular and independent of the BIOS itself, The control applet is responsible for setting up the pre-OS environment and loading necessary modules so that authentication resources can be used. This might include decompression, entry-point tables, IRQ re-mapping, etc.

3.1.1.4 Memory Management

There are no functions like MALLOC in the pre-OS environment. So the control applet is responsible for hiding the details of allocating memory to modules, buffers, etc. from the lower layers. It is expected that it would do this in a standards based manner.

3.1.1.5 API Marshalling

It is assumed that some user authentication devices will require more memory space than is typically available in the pre-OS "REAL MODE" environment. Switching to and from 32-bit mode is doable pre-OS and the control layer is responsible for orchestrating this. For example: converting a FAR CALL from 16-bit into 32-bit format. More details will be discussed in subsequent sections.

3.1.1.6 Traffic Control

The authentication devices and the storage device are shown as separate sub-systems in the architecture above. Thus, the control applet is responsible for making calls to the protected storage sub-system and sending the necessary information to the authentication sub-systems and vice-versa. It is not intended that the service providers themselves access protected storage. Although, there is nothing that prevents it in the architecture. However, in a multi-threaded environment, there also needs to be a resource manager to prevent access conflicts.

3.1.1.7 Other BIOS Issues

Since the BIOS writer is responsible for the control applet, he or she will be writing it to meet the specific needs of a certain BIOS. Thus the control applet may handle other BIOS specific issues as long as it is understood that the applet will not be portable across all platforms. To be compatible with the specification it needs only to make calls to the authentication and protected storage hardware through the APIs defined in this document.

3.1.1.8 API Extensions

The IPAA applet writer may decide to provide API extensions to reduce redundancy and complexity in their control code. These extensions should not be visible to or used by the other IPAA layers, as this will limit the portability of the code.

3.1.1.9 Device Enabling and Configuration

The BIOS normally does a lot of platform configuration before proceeding to load the operating system. Since IPAA authentication takes place in the "pre-boot" space, the control applet must take the responsibility of enabling and configuring the necessary resources to enable authentication.

3.1.2 Application Programming Interface (API) Layer

This layer and its support are the subject of this document. Specifically, the specification defines function names, calling convention, return convention, buffer structures, and executable module "export" structure.

This layer defines a minimal subset of high-level function calls needed for user authentication and storage. They are intended to work with all user authentication and storage devices known, regardless of the underlying technology. Efforts have been take to provide for the three known authentication models: local, smart subsystems, and remote authentication. But, It is important that the architecture also be extensible so that future user authentication considerations can be accommodated as well. This might be in the form of new technology, new function calls, or new usage models.

It is intended that this layer will eventually be compliant with or included in an authentication interface standard. Since much of the work of this specification has been done before external standards are available, it is intended that this interface will be brought to bodies like BioAPI and TCPA for continued work in defining pre-OS user authentication on PC Platforms.

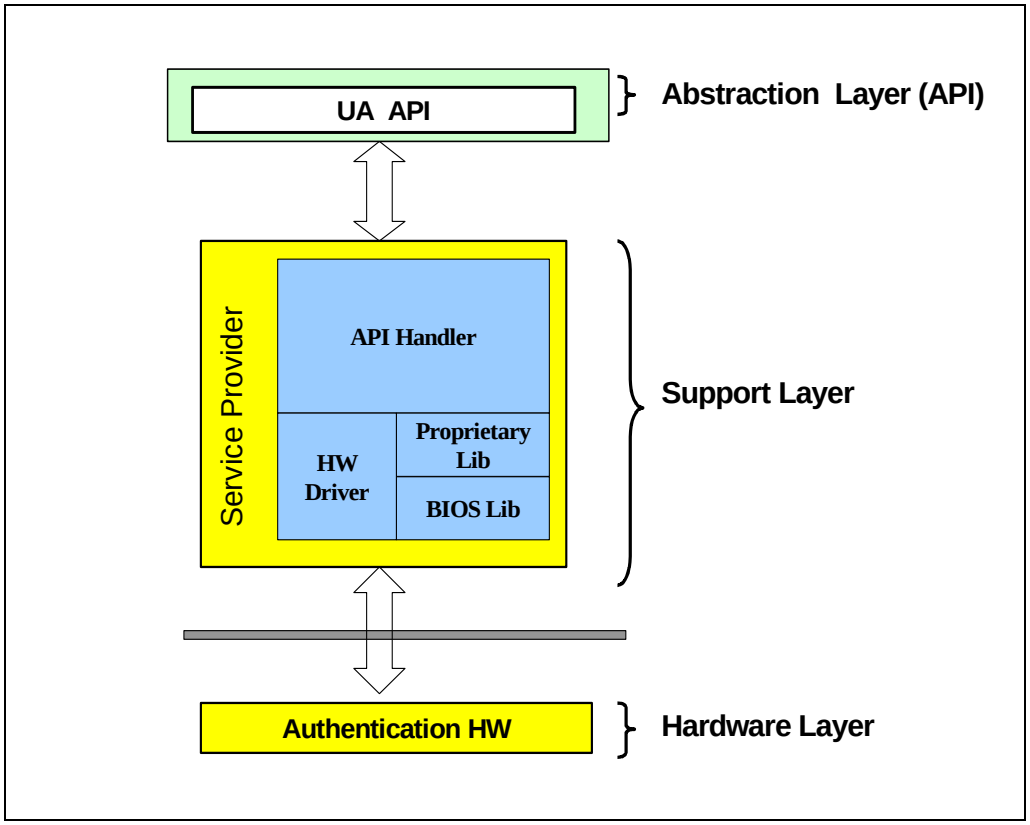
3.1.3 Support Layer

This layer, in the IPAA, is responsible for providing all of the device specific support. Whether it is an authentication or storage device. It must support the required functions delineated in this specification leaving the support of non-required functions up to the IHV, who would choose those functions that made sense for the device in question.

This layer is owned and developed by the authentication hardware manufacturer (IHV). It is basically responsible for translating the IPAA API calls into proprietary IHV, BIOS, and Device Library (LIB) calls. It is this layer that makes the IPAA API abstraction layer possible. Modules in this layer are referred to in this document as Service Providers (SP). It is thought that the one advantage of this architecture is that service providers, if coded to use the IPAA and standard BIOS interfaces only, would be portable across platforms. That is , the code would be tested and validated and would have higher confidence of running unchanged on other platforms thus easing the burden on platform developers, authentication developers, and BIOS developers.

Figure 2 shows a block diagram of a typical service provider. This is provided for discussion only as the actual internal architecture is up to the discretion of the IHV.

Figure 2: Service Provider Block Diagram



In this diagram there are essentially four components:

- API handlers and Logic
- Proprietary Libraries
- BIOS interface Libraries
- Hardware Driver

3.1.3.1 API Handlers and Logic

This block is the responsible for actually receiving the API function calls from the control applet and returning the appropriate information based on this specification. It does this by converting the API calls to the necessary sequence of calls to the vendor's proprietary library, BIOS standard interfaces, and to the hardware driver code.

3.1.3.1.1 Proprietary Libraries

This block contains on the vendor specific code that understands the specific algorithms for the particular device. It may interface to standard BIOS interfaces and the HW driver as well.

3.1.3.1.2 BIOS Interface Libraries

Although this block might not be present, it represents code that would expose standard BIOS services to the service provider as a whole. These may be certain bus interfaces (such as RS-232, etc.) or other services critical to operation of the device.

3.1.3.1.3 Hardware Driver

This block contains code that specifically knows how to communicate to the hardware device, which it exposes to the API handler and other blocks.

3.1.4 Hardware Layer

This layer represents that actual hardware that is on or attached to the platform for the purpose of user authentication. Devices are connected to the platform externally or internally via well know busses. External busses include; RS-232, Parallel port, USB, etc. Internal busses could include; LPC (Low Pin Count) bus, PCI (Peripheral Component Interconnect) bus, and PC-card bus.

The OEM decides which authentication devices, and busses, are supported and puts the appropriate service providers in the support layer. This architecture makes no distinction between built-in or external plug-in authentication devices at this time (although implementations need to take special security precautions to address replay attacks).

3.1.5 Non-Volatile Protected Storage

The IPAA architecture includes a protected storage interface to abstract the details of hardware that may be included on a system to provide the protected storage feature. This feature is vital to the role of IPAA as an asset and data theft protection mechanism, but enables other usage models as well.

Protected storage is essentially non-volatile storage that has a means of access control. This access control determines which entities (user, program, etc.) have permission to read, write, modify, update, etc., the information contained within the protected storage.

It is assumed that protected storage has some form of access control protocol that is used to protect against certain kinds of attack. This access control protocol is specified by name during the enumeration process so that the required protocols can be communicated to end-points such as the control or service layers. This method also allows for future protocols, although, only one protocol is defined in this specification. The protocol provides an anti-replay integrity channel thus ensuring the data hasn't been tampered with. Future protocols may provide data privacy as well. One encoding for the protocol is NONE. This allows the storage service provider to be totally responsible for implementing any special protocols required for the hardware.

Depending on the implementations of protected storage, it may or may not be available to both pre-OS and OS-present applications. It is intended that the interface to storage is necessary and sufficient for pre-OS access and at least a minimal subset of services that could/would be available OS-present.

The classes of information currently expected to be stored here include:

- User authentication data
- Device configuration data
- And possibly some global security parameters

Since this interface abstracts the details of the protected storage hardware it allows the control applet, and possibly the service providers, to access protected storage in a standard way that is portable across IPAA compliant systems. To do this, this specification makes some simplifying assumptions/requirements listed below:

- Storage is slot oriented
- Each slot contains a predefined header
- Each piece of data has a predefined header
- The actual data in the slot is meaningful only to the entity that created it
- The Storage component must be inextricably bound to the platform
- Protected Storage Service Provider hides the details of the underlying storage technology

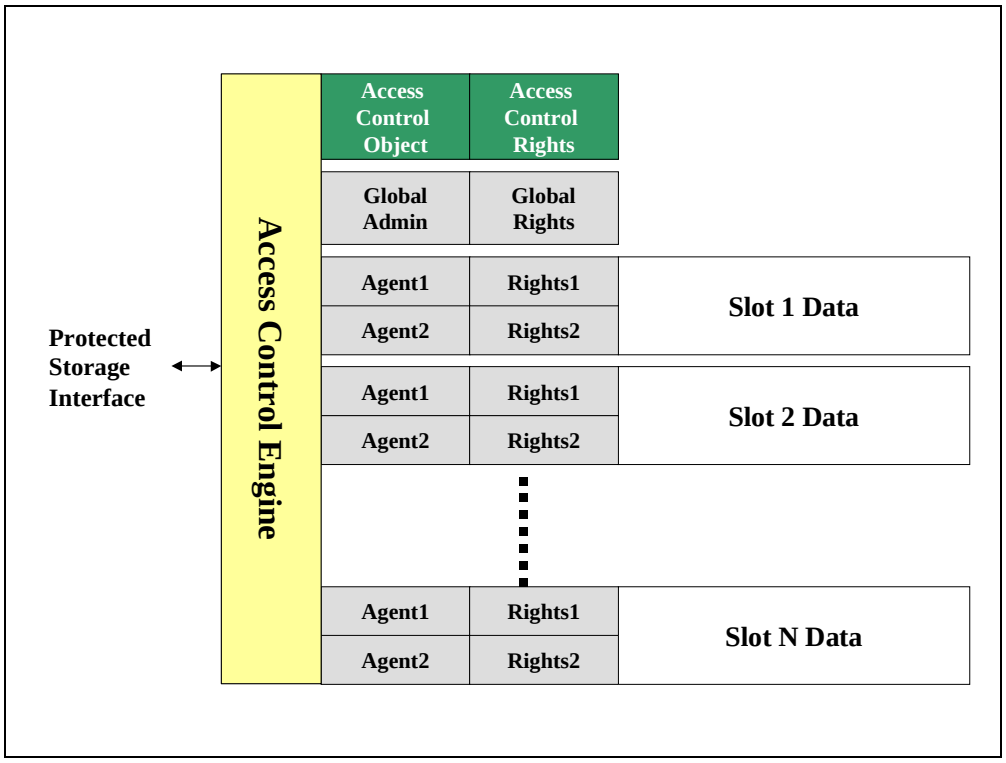
Some or all of the protected storage may be located off-platform, say on a smart card via an RS-232 connected reader, as long as it meets the expectations above. Additionally, to ensure *implementations* protect the storage, it should meet the following conditions:

- Initial state of storage which is equivalent to users not registered must be different from totally erased media (In this last case storage is damaged)
- Every storage state is tamper resistant, including the initial state. Tamper resistance, in this case, means that either unauthorized access to storage is reliably prevented with hardware, or changes to content can be reliably detected
- Data in storage should be resistant to interpretation by 3rd parties. Again, this means that either unauthorized access to storage is reliably prevented with hardware or the data is scrambled and virtually unreadable.

3.1.5.1.1 Protected Storage Architecture

In order to generate a high level interface, a high level model of protected storage was generated. As mentioned previously, it is assumed to be slot oriented as shown below in Figure 3. This was felt to be a rather natural assumption as there are many examples of this approach already (HDD sectors come to mind).

Figure 3: High-Level Protected Storage Architecture



The number of slots and the size of each slot are not specified in the specification, but can be determined from executing functions in the interface. The "global admin access control object" is the way that the platform owner or administrator authenticates themselves to the storage. Associated with the global access control object are global permissions. This permission object determines what the administrator is allowed to do within storage. With the proper set of admin permissions, storage can be "recovered," in cases where an employee may leave a company, but also prevents administrative access to the actual data in the slot.

For each slot there is assumed to be two (or more) access control objects. This allows for more secure control over data in the slot as the entities that have permission to write data into the slot can be different than the entities that can read data from the slot.

3.1.5.1.2 Permissions

The permission available to the administrator and slot are by design identical. It is presumed that a configuration utility that the owner runs would set the default settings that would be consistent with the owner's security policy. These permissions include:

- Read
- Write
- Allocate/Free
- Change Access Control Object (administrator can change ACOs in other slots)
- Reduce Permissions (administrator can also reduce permissions in other slots)

As mentioned previously, it is assumed that there is a configuration tool that is run by the "owner" of the "configuration manager" that pre-configures these permissions to a default value. After

that, function calls to change privileges can only lower, or reduce, the privileges that are present. For example: the global admin permission might be pre-configured to be "Allocate/FREE" only, allowing the administrator of the platform only the permission to erase slots.

3.1.5.1.3 Choice of Protected Storage

Protected storage can be implemented on a PC platform in many ways. This specification allows the OEM to choose which is right for their customer base and business goals. This may vary from careful implementation using common platform components (FLASH, Chipset, Boot code, etc.) all the way to sophisticated hardware subsystems that provide extra protection against additional classes of attacks. In all cases the protection provided should be enough to prevent software-based attacks yet at the same time allow for "recoverability", "serviceability", and "manageability (local and remote).

3.2 Pre-boot User Authentication Services Architectural Goals

The high level goals of the architecture are shown below. It is assumed that the some of these goals are implementation dependent and ultimately up to the discretion of the OEM/BIOS writer.

3.2.1 Required

- *Simple*: Provide a high-level abstraction layer between the controlling applet and hardware that can be used to authenticate users. A small number of simple functions is preferred.
- *Agnostic*: The architecture and implementations based on it must be BIOS, Token, and Bus independent while leaving room for OEM differentiation (Architecture must support Biometric and Smart Card technology).
- *Modul*ar: New modules can be "plugged" in by the OEM as deemed necessary without having to rewrite the User Authentication Applet.
- *Extensible:* Provides for future authentication technology and methodologies
- *Timely:* The architecture must support biometric and token-based authentication, in addition to other well-known authentication mechanisms.
- *Easy:* Pre-OS details and complexities of storage and memory allocation should be hidden from token stacks.

3.2.2 Recommended

- Authentication data can be used by authorized OS-present applications
- Support ATA-3 hard disk drive password
- Support for remote manageability (WOL, WOR) to get past pre-boot authentication
- Support policy configuration (BIOS SETUP, OS-present)
- Support enrollment of new users (Pre-OS, OS-present)
- Support OS-present management of enrolled templates
- Support hierarchical privilege levels

- Support re-authentication upon resume from power management states (ACPI:S3,S4,S5 and APM: STR, STD, SOFF)
- Protection of IPAA code is provided by the BIOS protection methods (None, Signed BIOS, etc.)
- Support PCXX, ACPI requirements
- Compatible with pre-OS BioAPI (when it becomes available)
- Compatible with PXE/BIS and other pre-OS BIOS initiatives

This document provides an interface and structure description of user authentication and storage API Libraries as components of a system-wide Pre-boot User Authentication and storage infrastructure and an interface to the BIOS Security Applet. These interfaces are available to pre-boot applications, such as a System BIOS and Setup, and are *not* available after the operating system has been loaded. It is assumed that if OS-present access to these services are to be made available, a separate OS-present API Layer Library and driver stack will be required to enable them.

3.2.3 BIOS Requirements of the Architecture

The Architecture needs to take special BIOS related issues into account. These issues are:

- Must support modular implementations. Multiple Tokens must be supported simultaneously by the system for multi-factor authentication and different media can be used as storage. Further it is assumed that number of Tokens available for authentication can be changed dynamically independently of system state – docked, connected to network etc. This means that components of IPAA system must be coded as self-encapsulated code modules and communicates with other modules and with conventional BIOS code through commonly defined interfaces.
- Interfaces must be BIOS independent. It is assumed that modules of Token and Storage Service Provider Libraries are designed and supplied by Token and Storage vendors. This means that these modules must be BIOS independent and compatible with any BIOS vendor.
- Must not preclude support for very large code modules. One of the groups of devices used as tokens is biometric devices such as Fingerprint sensors. Deriving of user templates from images captured by such devices typically is computationally very intensive process. As a result code size of supporting Token SPLs may be as large as 500K expanded and 50-60K compressed.
- Token enumeration shouldn't have significant impact on start-up time: When system supports multiple Tokens with large code size of SPLs it can be impossible to simultaneously load all of them for user authentication. Therefore the essential is the system's capability to enumerate available Tokens and registered Users without actual loading of Token SPLs. System may use Storage to register Users and Tokens and use this registration information to generate User prompts.
- Should not preclude Dynamic loading and linking of modules : When multiple Tokens are present in system User selection or system policy determine which Token is to be used for User authentication. After decision is made the selected Token SPL must be at run-time loaded into memory, linked and executed. It makes sense to expand this dynamic loading and linking capability onto all components of IPAA system and define corresponding common methods and data structures.
- Should not preclude execution in protected mode. Given the potentially large size of Token SPLs, the only possible resolution of pre-boot memory constraints is to move executable code into extended memory. Therefore the BIOS must be capable to supporting execution in protected mode. This is the preferred execution mode.

- Should not preclude execution in real mode. Given the fact that some authentication device support will be ported from current 16-bit code, and the fact that these devices may require a small amount of real mode resources, execution in real mode should not be precluded. Additionally there may be a need to switch back and forth between real mode and protected mode to access certain services on the platform.
- Support various forms of storage media for the Token Service Providers: A Flexible IPAA system assumes that the number of tokens required for user authentication depends on the system state. For example, as a result of a pre-boot network connection, the system policy may require additional user authentication. The Token SP for this authentication may be located on a network server and downloaded for execution (possibly using the PXE transport).
- Arch should allow for development in various development languages: Many equivalents of service providers are available today as OS-present drivers and applications. By supporting C and Assembly language, solutions could be provider more quickly. This means that the architecture needs to take the differences of the executable modules into account.

3.3 Execution Model

The typical amount of conventional memory, not occupied by BIOS at POST time and available for pre-OS security services, is between 100 and 300K. The amount of memory consumed by pre-OS security services can be much larger. The only way to overcome the real-mode memory constraints is to place the Token SPL in high memory and select either the big-real mode or the protected mode execution model.

Only the protected mode environment is usable as the big-real mode execution model allows operations on a huge data space and disallows operations of a huge code space. Thus of the two operating environments, 16-bit real and 32-bit protected modes, the latter is the most preferable. It simplifies module relocation, interrupt handling, and overall is more simplistic and consistent.

Thus, for the rest of this specification it is assumed that the IPAA system will execute in real and 32-bit protected modes only.

3.3.1 Development Tools

Many Tokens available today, are used to improve the security of various OS-present applications. These solutions already have supporting software coded as DLLs and Device Drivers. This software is frequently developed with some combination of Microsoft C, C++, or Assembly languages.

It is the intention of this specification to take advantage of these available code bases, by defining the various data structures, export mechanism, etc. to allow use of C as well as ASM for development. Because these development tools produce relatively different executable file formats, the IPAA implementation/Architecture take this into account.

The goal is to use any of the 16 and 32-bit C compilers and linkers. Further, it is assumed that the Token SPLs will be developed using the following Microsoft languages:

- x86 Assembly Language
- C Language
- A combination of the x86 Assembly Language and the C language

Tools of other vendors can also be used as soon as they produce compatible output file formats.

4.0 Interface

4.1 Data Transfer

Large amounts of data need to be transferred between different modules at run-time including templates and return values. To simplify this transfer all data will be placed into globally allocated buffers and "Far" pointers to these buffers will be used as parameters of the functions.

4.2 Parameter Format and Marshalling

Run-time calls to functions exported by service provider libraries originate from real mode BIOS and Setup. This means that the code doing the calls will use **Segment:Offset (**16-bit**:**16-bit**)** format for far pointer parameters.

At the same time, far pointers received by service providers executing in 32-bit protected mode, must have their far pointers in **Selector:Offset (**16-bit**:**32-bit**)** format. Thus, parameter marshalling is required.

In order to maintain proper alignment of 48-bit protected mode far pointers, they will be placed into a Quadword forming a **Selector:Offset (**32-bit**:**32-bit **)** value where the high word of the selector always equals zero.

Also, to reduce the complexity of handling parameters, IPAA-compliant systems will only use the following parameter formats in the function calls:

- Doubleword (32 bits)
- Doubleword real mode far pointer Segment:Offset (16-bit:16-bit)
- Quadword protected mode far pointer Selector:Offset (32-bit:32-bit)

4.3 Calling Conventions

To properly route function calls and perform necessary translation of input parameters, the following list is a set of conventions that support all interfaces:

- If any of functions are not supported, the module must return FUNCTION_NOT_SUPPORTED error code.
- All data is routed to and from functions through globally allocated buffers. Far pointers to these buffers are passed as function parameters.
- Return values from functions are always Doublewords.
- Functions return error codes as negative values in EAX register (bit 31 is set).
- A return value of '0' from a function indicates successful execution.
- All functions must return 32-bit values in EAX register, irrespectively of mode of execution (real or protected).
- If a function expects far pointer as parameter, and it is called with a NULL pointer value (0 value), it returns the positive size of the largest buffer required by the function. If the function expects more than one far pointer and is called with NULL value instead of one of them, it

returns positive size of the largest buffer for this pointer. If more than one NULL pointer is passed to function the return value is undetermined.

- Any buffers passed to a function, as a parameter should not contain other embedded far pointers that require marshalling.
- All functions support C run-time calling convention. That is, parameters are pushed onto the stack from right to left and the caller is responsible for stack recovery (C naming convention is irrelevant).
- All function parameters will be in the following format:

  DWORD Module Descriptor, DWORD Function Descriptor, VarArg (0 or more). Justification and description of File and Function Descriptors will be in the following sections.
- Interface must support **GetCapabilities** function with the following syntax:
- `GetCapabilities` (DWORD Module Descriptor, DWORD Function Descriptor, BYTE* fpBuffer)
- Functions are responsible for filling the buffer with descriptors of all functions supported by the module. See later sections.
- No assumptions are made regarding the preservation of register values except for the following:
  - Functions will preserve registers that have to be preserved by any C program: ESI, EDI, EBP, and Direction Flag. Functions will additionally preserve GS and FS register values. They are reserved for system-wide purposes.
  - Kernel of IPAA system changes only format of function arguments (argument marshalling) but doesn't change number of arguments.

4.4 Descriptors

4.4.1 Module Descriptor

Module Descriptor shown below is a Doubleword value that is used at different phases of execution and contains all the necessary information to:

- Identify module functionality (decompression file, data file, toke file, etc.)
- Identify type of information is associated with module protected storage slot (device configuration, user authentication templates etc.)
- Identify type of media the module is stored on (FLASH, HDD, network etc.)
- Identify device to use during the function call.

At every phase of execution only part of information included in Module Descriptor is actually used. Other parts can be ignored or skipped.

For example when Module Descriptor is passed to the loader, it uses the Media byte to select which media specific Read/Write routines need to be run. It uses the module descriptor byte to find the requested module on target media. In this particular case, the loader ignores all other information in the Module Descriptor.

The information in the Module Descriptor is required for function execution. Optional fields are noted in the description of the various functions. The format of Module Descriptor is shown below in Table 2.

Table 2: Module Descriptor Format

Storage byte		Media byte		Module descriptor byte		Device byte	
Storage class bits	Storage number bits	Medium class bits	Medium number bits	Module class bits	Module number bits	Interface number bits	Device number bits
31 – 28	27 – 24	23 – 20	19 – 16	15 – 12	11 – 8	7 – 4	3 – 0

- Storage byte

The Storage byte is used as identifier of information stored in the slot. It contains two sub-fields; Storage class (4 bits), and storage number (4 bits).

Storage Class: specifies the type of stored information and type of access. Format of Storage Class is shown below in Table 3.

Table 3: Storage Class Nibble

Bit number			
31	30	29	28
Class category			Access type
0 – 7			0, 1

Access type: specifies single-slot or multi-slot access – see note in section 5.1

0 – Multi-slot access.

1 – Single-slot access.

Class category: sequential number. Next categories are assigned.

0 – Authentication.

1 – Messaging interface

2 – BIOS interface

3 – 7 Reserved

Class category and Access type together yield next Storage classes.

0 – Authentication class, configuration data slot – contains information necessary for configuration and initialization of authentication devices.

1– Authentication class, user authentication data slot – contains templates of registered users.

2 – OS messaging interface class, multi-slot access.

3 – OS messaging interface class, single-slot access.

Etc.

Storage Number: is a sequential number that refers to the storage slot within the storage class.

- Medium byte

  The Medium byte is used as an identifier of the medium where the module is located. It too consists of two 4-bit fields. The Medium Class and Medium Number fields.

  Medium Class: specifies type of the medium. This is currently defined as:

  0 – Flash Part

  1 – HDD

  2 – Smart Card

  3 – Network

  4-15 *Reserved*

  Medium number: is a sequential number for a given media device (i.e. disk 1, disk 2 etc.)

- Module descriptor byte

  The Module descriptor byte is made up of two 4-bit fields. The Module Class and Module Number fields.

  Module Class: specifies the module's functionality.

  0 – decompression module – performs decompression of loaded module into memory for execution.

  1 – core module – responsible for transitions to and from protected 32-bit mode, interrupts handling and parameter marshalling.

  2 – storage SPL – supports protected storage hardware.

  3-7 *Reserved*

  8 – Token SPL – used to communicate with Token Device to perform authentication specific operations.

  9-15 Reserved

  Module number: is a sequential number for a given class.

- Device byte

The device byte is used to convey the "number" of the device to be selected. That is, if there are multiple devices on the platform, this byte indicates which device the function was intended for. In such case this byte serves as function modifier and allows the caller to select the target of the call – generic or device specific.

The next description assumes that all devices supported by module are enumerated in next way – first all interfaces are enumerated sequentially starting from 1 and then all devices attached to the same interface are also enumerated sequentially starting from 1. Finally common device number consists from interface number and number of device itself.

Interface number: specifies number of the interface.

0 – reserved value

1–N – number of interface

Device number: specifies number of device attached to interface

0 – reserved value

1–N – number of device

The type of information returned by the function that uses this byte is dependent on the encoding used in the field. The differences are described below:

Device byte value: specifies the target of call – generic or device specific call.

00 – The called function is used to convey/return generic (not device specific) information at library level.

X0 – The called function is used to specify/return information for or communicate with the specific interface controller numbered X.

XY – The called function is used to specify/return information for or communicate with the specific device numbered Y attached to interface X.

0Y – Invalid value.

4.4.2 Function Descriptor

The Function Descriptor informs the service provider which function to execute, the number of parameters in the buffer, and which ones are Doublewords vs. Far Pointers. The latter two are used for parameter marshalling.

The format of the Function Descriptor is shown below in Table 4

Table 4: Function Descriptor Format (Bit Definitions)

Argument Mask																Number of Arguments	Sub-function Number	Function Number
Word																Nibble	Nibble	Byte
31															16	15 12	11 8	7 0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0			

- **Function Number**.This is sequential number assigned to function within function category.
- **Sub-function Number**. This is additional function specifier.
- **Number of Arguments**. This is number of arguments passed or requested by function.
- **Argument mask** . This word has a 16-bit mask. Each bit represents a parameter being passed as part of the function call. A ‘0’ indicates that the corresponding parameter is a Double word and a ‘1’ indicates that the corresponding parameter is a Far Pointer. Bit 16 represents the 1st parameter, bit 17 the second, and so on. Bit 16 and 17 are always set to ‘0’ because the first two parameters are always the Module and Function descriptors which are Doublewords.

4.4.3 Use of module and function descriptors in Storage and Token SPL

Every Storage and Token SPL function receives a Module and Function descriptor as the 1st and 2nd parameters in a call.

All information but the device byte contained in Module Descriptor may be ignored by functions. This information is used only by the IPAA kernel to properly identify target of call.

Device byte can be interpreted by functions. Generally this interpretation is performed in the next way.

- If the function is called with Device byte equal to 0 such call is considered to be relevant to entire library i.e. to be made at library level;
- If the function is called with Device byte not equal to 0 such call is considered to be relevant to device or interface controller i.e. to be made at device level.

It is assumed that during initialization call the library enumerates internally all interfaces and all devices and assigns them sequential numbers starting from 1. Resulting numbers can be used as device and interface handles.

For instance value 0x21 in device byte can be interpreted as device 1 attached to interface 2. When functions are supported at Device level the Device byte is used to convey to function these device numbers.

Functions need only interpret the "function and sub-function number fields" contained in the three least significant nibbles of Function Descriptor. The rest of the information serves the same purpose of mode transition and can be ignored by SPLs.

4.4.4 Rule to construct Device Byte

To call function at device level the value set into Device byte must be constructed as next. The valid interface number must be set from 1 to maximum returned by "xxGetInfoFn" call as shown in Table 27 and Table 61. If there is only one device supported and attached to interface – the function must ignore lower nibble of Device byte. If more than one device is attached to interface and such attachment is supported – the lower nibble must contain valid device number as shown in Table 30 and Table 62.

4.5 Exporting Functions from Modules

One of the most widely used methods of calling BIOS de-coupled services is to call them through exported entry points. Just one exported entry point is enough to call all exported functions since the function number of interest is passed as part of the function descriptor. This method of exporting functionality in modules will be assumed in the rest of this document.

Table 5 defines the export structure for locating and validating the module (service provider) entry point.

Table 5: Module Export Structure

Name	Module or Vendor Name	Length in bytes	Checksum	Entry Point Offset	RVA	Attribute
Size	16 Bytes	Byte	Byte	Doubleword	Doubleword	Word
Value	Vary	28	Vary	Vary	Vary	Vary

- Module or Vendor Name – A vendor specific name used to identify the module. The same name is used as part of the storage slot header. Those slots will be later associated with this module name. More about this later.
- Length – The length of this structure in bytes. Used for Structure validation.

- Checksum – The 2's complement of byte-wide checksum of all other Structure fields. Used for Structure validation and reliability.
- Offset – The offset of the module's entry point relative to the beginning of the file..
- RVA – A constant value to be added to all internal offsets at run time. More about this below.
- Attribute – Module attribute word.

  Bit 0 … 1 = module is 32-bit, 0= module is 16-bit

  Bit 1 … 1 =execute in protected mode, 0 = execute in real mode.

The caller will validate the Structure by performing byte-wide summing of Structure fields on a specified length. By definition this sum must yield 0 value. This checksum is not intended for security.

This structure can be generated and populated at design time. Appendix B contains an example of assembly language code that accomplishes this.

4.5.1 RVA field

Relative Virtual Address (RVA) is a value to compensate for differences in output file formats produced by C and Assembly language linkers.

If source code for 32-bit protected mode module is developed in Assembly language and compiled into OMF object format , it can be linked using 32-bit linker from a MASM package. The output file can then be converted into binary format using the EXE2BIN program. The resulting binary image will have all internal offsets relative to the beginning of the module (it is assumed that the code segment is placed into module first). In this case the module has no displacement to compensate and the RVA field must be set to a value of zero.

If source code for 32-bit protected mode module is developed with a C language (or combination of Assembly and C) it must be compiled into COFF object format and linked using a 32-bit linker from the C/C++ package. The output file from this operation contains a PE header and cannot be converted into binary form using EXE2BIN. The PE header can be stripped out using binary file modification tools, but the resulting binary image will still have all internal offsets relative to the beginning of PE header. To compensate for this displacement the RVA field must be set to the size of the stripped PE header. The service provider programmer can take special steps to ensure that the PE header is always equal to 4 Kbytes. Thus the RVA value can be fixed at 4 Kbytes (1000h).

See also Section 3.3 for additional information.

4.5.2 Storage and Token SPL Module Structure

It is assumed that a module starts from the beginning of its export structure. That is, the 0 offset is located at the beginning of the file. Contents of the Export Structure are specified above.

5.0 Storage Service Provider Library

5.1 Organization of Storage

To simplify housekeeping, protected storage consists of logical **"**Slots**"** of equal size. Internally, the physical storage mechanism can be implemented either as a homogeneous or discreet medium. The difference between the two is in terms of granularity. In the latter case, it is the responsibility of the storage service provider to map the logical slots to the physical equivalent of the medium.

A slot is considered to consist of two logically separate areas – a data area and a name area. All slot name areas combined together and all slot data areas combined together can be referred as to "Storage name space" and "Storage data space" respectively. The Storage data space always map onto a "real" medium such as FLASH or hard disk drive. But, the Storage name space may be "real" or "virtual". That is, it may be a real area set aside inside the storage medium, or it may be some media specific access mechanism, such as the way file names are used to map to data in files.

Table 6 below shows the logical view of storage layout.

Table 6: Logical Storage Layout

Space	Area	Comments
Name Space	Storage Name Header [1]	Vendor ID, Storage metrix
	Storage Security Header [2]	Storage-wide Access Control Values
	Slot 0 Name Area [3]	
	…	
	Slot N Name Area [3]	
	Storage Security Trailer [4]	
Data Space	Slot 0 Data Area [5]	
	…	
	Slot N Data Area [5]	

NOTES:

1. Storage Name header contains all storage identification and specific parameter values such Vendor ID, slot count, slot size, media size, etc.
2. Storage security header contains all necessary objects which support authorized access to protected storage. This access is intended to be used by OEMs, IT personnel, etc. Security Objects consist of Access Control Objects and Access Rights. The exact nature of the security objects depends on the security protocol accepted by the storage vendor. See the description of protocol below. This header is optional and is not present if access to storage is not controlled by a security protocol.
3. All Slot Names areas in order.
4. The security trailer is a security object which is used to validate the integrity or authenticity of the storage name space. This area is optional. If it is present, it is defined by the protocol used.
5. All Slot Data areas in order.

Table 7 below shows the logical layout of each slot.

Table 7: Logical Slot Layout

Slot Area	Slot Header	Field Size	Field Name	Comments
Name Area	Name Header	16 Bytes	Slot Name [1]	
		Doubleword	Slot Descriptor [2]	
	Security Header	Vary	Access control values[3]	Fields in this header, and their use, are defined by the actual security protocol supported by the storage subsystem.
Data Area	Data	Vary	Data	
	Security Trailer	Vary	Integrity Control Value [4]	

NOTES:

1. All slots that are owned by an application like Token SPL, BIOS, etc. – have the same name. This way slots are clustered together and allow group access.
2. Slot descriptor follows Module descriptor format described in Table 2 and is unique throughout the system.
3. Access Control Values may include nonces, pass-phrases, access keys, etc. and are specific to the security protcol chosen.
4. Integrity Control Value is Security object which is used to ensure integrity of Slot Data area. This field is optional and protocol dependent. This field may also contain objects like “signed response”, etc.

Table 8 below shows an example of slot usage by two different applications – ABC Token and DEF Token. Security Headers and Trailers are intentionally not shown. This example does not exclude multiple configuration slots.

Table 8: Example of Slot Allocation

Application	Fields		Comment
ABC Token	Name	ABC Token	Configuration slot of ABC Token #0
	Descriptor	00xx80xx	
	Data	Data	
	Name	ABC Token	Authentication slot #0 of ABC Token #0
	Descriptor	10xx80xx	
	Data	Data	
	Name	ABC Token	Authentication slot #1 of ABC Token #0
	Descriptor	11xx80xx	
	Data	Data	
DEF Token	Name	DEF Token	Configuration slot of DEF Token #1
	Descriptor	00xx81xx	
	Data	Data	
	Name	DEF Token	Authentication slot #0 of DEF Token #1
	Descriptor	00xx80xx	
	Data	Data	

It is further assumed that data in slots are allocated in sections where every section has the format as shown in Table 9. There may be one piece of data in a slot or many. Each has the format as shown.

Table 9: Data Section Format

Field Size	Field Description	Comments
Word	Size of data in section [1]	
Word	Section tag [2]	
Vary	Data	

NOTES:

1. This value excludes size of 'Size' field itself and size of Tag field i.e. Data Size = Total Section Size - 4.
2. Section tag is identifier of type of data in section. Examples are User Template, Device configuration data, ATA 3 Password etc.
3. Organizing data into sections provides browsing capability for assocated management software. It is assumed that the first section in a slot begins from the 0 offset of slot data area and that this exact position is occupied by the section size. A value that allows the offset of the beginning of next section to be computed. The next word, after the size word, is the section tag.
4. A section is considered to be last one in a slot if any one of two conditions are met:
   - The Doubleword at the offset where a section is expected equals '0' (Tag plus Size fields)
   - The offset where a section is expected is beyond slot length.

Note: About tags and size fields.

Generally all size fields in all places throughout this specification have there own size specified by a doubleword. This is done to keep doubleword alignment of data where ever possible, though a doubleword is never required by size of data itself. In cases when the specification deals with data allocated in sections and the sections have to be "tagged" – the word for placement of the tag is "borrowed" from the "size" doubleword. In this way, data aligment is maintained. The software that browses such sections must mask-off the tags to obtain the size of data.

Note: About scanning of data in slots and buffers

Browsing software uses tags and size fields in order to browse sections in slots or buffers. It is assumed that the first section always starts from zero offset of a slot or buffer data area. Browsing software reads the first doubleword of a section and masks-off the tag in order to extract length of data. The next section will start at the position specified by offset: "Current Section Offset + Current Section Data Length + 4". The tag provides information about section content. Browsing stops when both the data length and the tag of the next section are zero or the obtained offset is beyond the size of the slot or buffer. In other words – the zero doubleword is an "end-of-scan" marker. This also means that sections with zero data length must have non-zero tags to avoid premature end of scan.

Note: About classification of tags.

All tags assumed by this specification are described in Appendix E. One of the assigned tags – zero tag, – has special meaning of "no tag" or "not specified tag". It can be used everywhere where tags are used assuming the limitation above. Generally, if the slot or buffer contains sections with different size and nature of data – all sections must have non-zero tags. Zero tags therefore have practical usage only in slots and buffers with homogenious content. That is, when all sections have the same size and data nature such as User Authentication Data slots with user templates.

Note: Thee are two different methods defined for allocating sections of data in slots.

- Single Slot Allocation. This method is used with small data sections so that multiple sections fit into the same slot. With this method, each section must completely fit into a slot. Expansion of a section

data past the slot boundary is disallowed. The distinguishing feature of this allocation method is the potential to manipulate data inside the slot. Note then that a subset of this method is the allocation of single section in single slot.

- Multi-slot Allocation. This method is used with large sections that span over multiple slots. When data is written into a slot with this method, the slot boundaries are disregarded. As a result, it is impossible to manipulate data inside the slots without first concatenating them in memory.

Note: About initial media state

The "initial" or "erased" state of any medium byte, as it is returned by interface functions, is assumed to be 0. It is the responsibility of the interface code to perform the necessary inverse (NOT) operation to satisfy this assumption.

5.2 Access Control

Some of the functions of the storage interface are protected in that that they allow only authorized access to storage content. The method of access control is storage specific and is defined by the supported security protocol. The security protocol also defines all required objects as well as each step of the communication flow between "end-points". One end-point is the storage hardware or its support stack, the other end-point is the requester of data. In many implementations the requester is the BIOS security applet but the architecture doesn't preclude the token service providers themselves being endpoints.

Storage can employ a variety of security protocols. But it is assumed, in this specification, that protected storage hardware on a platform will only support one protocol at any given time. And that this protocol probably won't change over the lifetime of the platform.

All information regarding the supported security protocols, including types of security objects, cryptographic methods, sizes, etc. is returned by `stGetInfoFn` function (see Section 5.4 below).

As a starting point, this specification considers two different protocols - None and IPAA "Anti-replay Integrity Channel #1"(AIC#1). These two protocols are discussed in more detail in section 7.0. Storage vendors may define other protocols at a later date, but their encoding in stGetInfoFn needs to be defined and agreed to by a IPAA standards body (to be defined).

The "None" protocol is a special security protocol which allows the storage service provider itself to deal with access control. One example of this might be the use of a smart card as external protected storage. The service provider knows either a shared secret or has public key credentials that can be used to either "unlock" the card or to cause it to yield "slot" contents. Another example could be the use of system FLASH such as Intel's Firmware Hub. In that latter case, though access to the storage medium is not controlled, it can be "locked down" by BIOS after use so that OS present access to data is not possible.

Regardless of the protocol, except the "none" protocol, it is assumed that:

- Storage has one "access control object" that controls access to storage as a whole. This Object is assigned index 0.
- Each slot has one or more "access control objects" that control access to the individual slots. These objects are assigned indexes 1,2 to N.
- Each of the access control objects above have permissions associated with them. These "access rights" are declared in a Doubleword-size bit-mask. These access rights control all operations that are available to the various "protected" functions. Functions that operate on

access rights only allow the reduction of rights– i.e. access rights can be de-asserted but cannot be asserted.

- The storage-wide Access Control Object and associated permissions are stored in the Storage Security Header – see Table 6. Slot related Access Control Objects and associated permissions are stored in Slot Security Headers – see Table 7.

Table 10 below shows assignment of indexes to Access Control Objects and Table 11 shows the assignment of permissions to bits in the permissions Doubleword (Access Rights bit-mask). The user and administrative permissions are symmetrical in the architecture allowing implementations to determine the actual default "rights" as needed by various regions, markets, etc.

Table 10: Mapping of Access Control Objects Indexes

ACO #	Use	Comment
0	Storage administration	Grants access to any slot
1-N	Slot administration	Grants access to associated slot with permissions associated with ACO#1– ACO#N

Table 11: Access Rights Bit Definitions

Bit number	Bit value	Associated Right	Comment
0	0	Read data from associated slot.	"Associated" slot means that ACO controls operations on slot where it is defined.
1	0	Write data into associated slot.	
2	0	Free associated slot.	
3	0	Lock/Unlock associated slot	
4 – 7	0	Reserved	
8	0	Set new value of associated access control object.	
9	0	Reduce associated Access Rights.	
10 – 15		Reserved	
16	0	Administrate – read data from other slots.	Administrate permissions allow access and modification to other slots and storage as a whole.
17	0	Administrate – write data into other slots.	
18	0	Administrate – allocate/free other slots.	
19	0	Lock/Unlock other slot or Storage.	
20 – 23	0	Reserved	
24	0	Administrate – set new value of Access Object on other slots or Storage	
25	0	Administrate – reduce Access Rights in other slots or Storage.	
26 – 29	0	Reserved.	
30	0	Reserved	
31	0	Reserved	

The tables shown below show encoding for various fields found in the stGetInfoFn return buffer – see Section 5.4.

Table 12: Protocol Type Field Definitions

This field defines the mechanism that will be used to communicate the security protocol at the programming interface to the storage sub-system.

Value	Protocol Type	Comment
0	None	No security protocol exposed at the API. Data protection is ensured by the storage service itself.
1	IPAA AIC#1	
2 – 0x7fff	Reserved.	
0x8000	Vendor specific[1]	

Table 13: Security Header Format for the "AIC#1" Prototocol

This table defines the fields necessary to support the AIC#1 protocol along with their various sizes. This will be useful in later sections in understanding the size of the overall header buffer needed to communicate with the storage service libraries.

Structure Size	Structure Name	Field Size	Field Name	Comments
4 bytes + Sizeof (Protection Challenge object)	Security Header	Vary	Protection Challenge Object [1]	
		Doubleword	Access Credential Index [2]	

NOTES:

1. Protection Challenge Object may have different nature depending on the security protocol used.
2. Access Index is the sequential number of the Access Control Object used to access a slot.

Table 14: Security Header Format for the "None" Prototocol

This table is put here to emphasize that the "None" protocol indicates that a security protocol is not communicated across the programming interface. In later sections that refer to the security header, the length is zero (or equivalent).

Structure Size	Structure Name	Field Size	Field Name	Comments
0 Bytes	Security Header	None		

NOTES:

[1] The use of the phrase "vendor specific" here and throughout the rest of this document indicates a value of field that is intended to define values not currently supported. This capability is intended for test purposes only.

Table 15: Hash Type Bit Definitions

Value	Hash Type	Comment
0	None	
1	MD3	Included for compatibility
2	MD4	Included for compatibility
3	MD5	
4	SHA1	
5 – 07fff	Reserved.	
0x8000	Vendor specific	

Table 16: Symmetric Encryption Type Bit Definitions

Value	Encryption Type	Comment
0	None	
1	DES	
2	3DES	
3	AES	
4 – 0x7fff	Reserved.	
0x8000	Vendor specific	

Table 17: Access Control Object Type Bit Definitions

Value	ACO Type	Comment
0	None	
1	Pass – phrase.	
2 – 0x7fff	Reserved.	
0x8000	Vendor specific	

Table 18: Access Control Object Encryption Type Bit Definitions

Specifies the method to protect the ACO when it is being changed via a command from an authorized end-point.

Value	ACO Type	Comment
0	None	
1	SHA-1 Hash-based one time pad	See appendix for definition
2 – 0x7fff	Reserved.	
0x8000	Vendor specific	

Table 19: Protection Challenge Object Type Bit Definitions

Value	Challenge Object Type	Comment
0	None	
1	Nonce	A random number used only once
2 – 0x7fff	Reserved.	
0x8000	Vendor specific	

Table 20: Security Trailer Type Bit Definitions

Value	Security Trailer Type	Comment
0	None	
1	Hash.	
2 – 0x7fff	Reserved.	
0x8000	Vendor specific	

Table 21: Storage and Slots Lock/Unlock Bit Definitions

Bit number	Bit value	Supported Lock/Unlock Type	Comment
0	1	Storage - Read	Storage can be locked/unlocked for reads
1	1	Storage - Write	Storage can be locked/unlocked for writes
2	1	Storage - Master	Storage can be master locked/unlocked. That is freeze the lock capabilities until a power cycle of the storage hardware.
3 – 7		Reserved	
8	1	Slot - Read	Slots can be individually locked/unlocked for reads
9	1	Slot - Write	Slots can be individually locked/unlocked for writes
10	1	Slot - Master	Slots can be individually master locked/unlocked
11– 15		Reserved	

5.3 Interface

All Storage interface functions can be subdivided on the following groups.

- Information functions provide information on storage and module interface.
- Initialization functions are used to initialize hardware.
- Enumeration functions allow caller to access name space and retrieve information about slots.
- Allocation functions allow caller to allocate new slots and free existing ones.
- Administration functions allow caller to perform administrative tasks such as change access credential, reduce permissions, etc.
- Data access functions allow caller to read and write data in slot.

Additionally, storage interface functions can be classified as protected and non-protected. Protected functions use mechanisms that allow only authorized access to the function capabilities. Non-protected functions allow free access to capabilities provided by the function (such as slot information).

Protected functions include Administration, Allocation and Data access groups, all other groups of functions are non-protected.

5.3.1 Structure of Buffers

All function types mentioned above use pointers to buffers in memory as a means to pass and return data. The buffer types and layouts are discussed below. Generally there are three types of buffers.

- Generic Buffer
- Header Buffer
- Data Buffer

The generic buffer is used by storage related functions. Both the Header and Data buffers are used by allocation and data Access functions.

5.3.1.1 Generic Buffer

The structure of a generic buffer is specified in Table 22. It is designed to support generic data interchange between the caller and the function.

Table 22: Generic Buffer Format

Field Size	Field Description	Comments
Doubleword	Size of data in this buffer excluding this size field [1]	Input & output
Vary	Data	

NOTES:
1. Caller must set this field to the size of allocated buffer less 4 (size of Size of Data field itself). On return function will update this value to the size of actually returned data.

5.3.1.2 Header Buffer

The structure of the header buffer is specified in Table 23. It is designed to pass all parameters of requested slot to the function. If any data must be returned from the slot it is returned in the Data Buffer.

Table 23: Header Buffer Format

Structure Size	Structure Name	Field Size	Field Name	Comments
20 bytes	Name Header	16 Bytes	Slot Name [1]	Input & output as specified in each function.
		Doubleword	Slot Descriptor [2]	
Vary	Security Header	Vary	Protection Challenge Object	
12 bytes	Parameter	Doubleword	Handle [3]	
		Doubleword	Offset [4]	
		Doubleword	Size [5]	

NOTES:

1. Slot Name is a free-form string that matches the name of the slot owner. This name is not unique – instead it clusters together all slots belonging to specific application. Slot name can be used by an application to identify its own slots.
2. Slot descriptor is unique doubleword identifier intended to be used by the system. Descriptor fields match Module Descriptor format and allow the system to classify slots by application and by type of data stored in them.
3. Handle is internal identifier used by the Storage SPL and is not interpreted by the calling application. Internal logic of Handle is incremental by nature. By incrementing the handle, the storage SPL can internally enumerate all slots in storage. Rules for use of handles are specified in later sections. All slot related functions return a handle that generally must be used as an input parameter for the next call to the same slot.
4. Offset is the slot offset of data to operate upon. It is used by data access functions. All enumeration functions neither use or change this offset. Any value set in this field by the caller is preserved by these functions.
5. Size of data to operate upon. It is used by data access functions and is preserved by all others.

Security header is present only if required by the protocol. Presence or absence of a security header and its fields is determined from the output of `stGetInfoFn` Function.

- Security header is not present for “None” protocol – see Section 7.2 if data protection in Storage is ensured by Storage locking. Otherwise it must be present what is indicated by size of ACO not equal zero.
- Protection Challenge object field is present if its size is not zero – see Table 29.
- Access Credential Index and Access Rights fields are present always if security header is present and protocol is other than “None”.

5.3.1.3 Data Buffer

The data buffer is used by data access functions only. Its structure is specified in Table 24. The size of this buffer must not be less than the value set in the "Size" field of the associated Header buffer.

Table 24: Data Buffer Format

Field Size	Field Description	Comments
Vary	Data [1]	Input & output
Vary	Security Trailer [2]	Input & output

NOTES:

1. Data size must be equal to value specified in Size field in the associated header buffer.
2. Security Trailer is the Security Object which is constructed by Caller (or Function) based on content of all buffers used in the function call and Access Control Object, selected to query access to slot . A storage SPL will verify data integrity by reconstructing the Security Trailer and comparing it with one passed by Caller. Security Trailer size is not included in 'Size" field of Header buffer. Security Trailer is used for validation purposes only and is not written into slot.

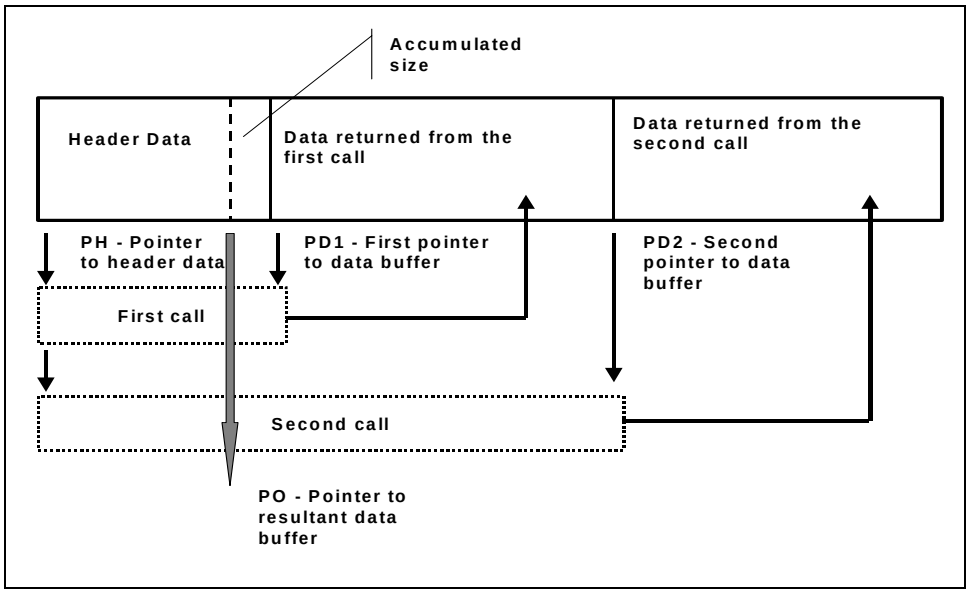
The security trailer field is optional and may not be used by the selected protocol. Presence of this field is determined by examining the output of the `stGetInfoFn` Function.

- Security Trailer field is present if its size is not zero – see Table 29.

5.3.2 Use of Buffers

The split buffer structure used in Data Access category of functions has a special advantage. Specifically it allows iterative calls to be made using the same function and header buffer but with different data buffers. This allows data to be retrieved from multiple slots efficiently. This principle is illustrated below in Figure 4.

Figure 4: Iterative Function Calls



The caller first allocates the large buffer with cumulative size equal to size of header and all data slots that have to be read. Then the caller issues the first call to a data access function using the pointers PH (header data) and PD1 (first data area of common buffer) as arguments. Data from the first slot is returned in the PD1 data buffer.

Then the caller issues a second call to access the next slot's data using the pointers PH and PD2 (second data area of common buffer) as parameters. Data from the second slot is returned in the PD2 data buffer and is effectively concatenated with data in buffer PD1.

Lastly, the caller sets the size field in the header to the combined size of all read slots. PO – is the resultant pointer to the combined data buffer that can be used as an argument for any call to the Token service provider libraries. Using the PO pointer as an argument automatically strips out the unnecessary header buffer for the Token SPL call.

5.3.3 Use of handles

Generally all slot related functions return a "handle" that must be used as an input parameter for subsequent calls to the same slot. The following are specific perturbations on the generation and use of handles.

- Every enumeration function which name contains "GetFirst**"** returns a handle that must be used as an input parameter for function containing "GetNext" (see following sections).
- Every enumeration function whose name contains "GetNext" can be called repeatedly and the returned handle must be used as an input parameter for next call.
- All enumeration functions return handles that must be used as input parameters for data access functions.
- Functions whose names contain "Read" don't change the handle (see later sections). This means that if only these functions are used, the handle can be stored by the caller internally and used for subsequent data access calls.
- Functions which names contain "Write" may dynamically change handle. This means that caller cannot save it for repeated calls.

5.3.4 Use of Anti-replay Objects

Use of these objects depends on the protocol supported by storage interface. An example of such an object and its use will be considered in Section 7.0 (applicable to IPAA AIC#1 Protocol).

5.3.5 Use of Security Trailers

The Security Trailer is the object that contains information that, in its initial incarnation, is used to validate the integrity or authenticity of the responding end-point. In the case of the IPAA AIC#1 protocol, this field will contain a keyed hash computed across all necessary fields in the header and data buffers. More details of this algorithm can be found in Section 7.0.

It should be pointed out that the fields and content of the security trailer object is defined by the security protocol supported by the system (more than likely defined by the protected storage on the platform).

5.4 Information Functions

Function 0x00

`stGetCapabilitiesFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor, fpByte buffer)***

This function uses a generic buffer with the layout specified in Table 25.

Table 25: Buffer of stGetCapabilitiesFn Function

Size	Name	Comment
Doubleword	Size of data in this buffer in Bytes excluding size field	Input & output
Doubleword	Fully qualified Function Descriptor of the First supported function	Output
Doubleword	Fully qualified Function Descriptor of the Second supported function	Output
…	Etc. for all supported functions	Output

NOTES:
1. Function descriptor for this function is 0x00043000.

Description of Function

`stGetCapabilitiesFn` function provides information about the Storage SP interface – function descriptors of all supported functions. For uniformity, this function returns information about itself as well. This function is required. Format and order of all parameters must be strictly followed. This function can be called before initialization of the Storage Library.

Arguments

When called with the Device byte equal to 0, the function returns descriptors of functions implemented at the library level. When called with the Device byte not equal to 0, the function returns descriptors of functions implemented at the device level. In general, implementation of any of functions at the device level is optional including `stGetCapabilitiesFn`. If not implemented at the device level, the function must return error. This also means that other functions are also not supported at device level.

It is also assumed that the same set of functions is supported for all installed devices and thus output of `stGetCapabilitiesFn` is the same for any of the non-zero device byte values.

Buffer Fields

- Input values:
  - Size field must be set to the length of allocated buffer less 4 (length of size field itself).
- Output values:
  - Buffer is filled with Function Descriptors of all supported functions and sub-functions.
  - Size field is set to the length of actually allocated data.

Error Codes

STO_FUNCTION_NOT_SUPPORTED	May be returned only if function is not implemented on device level.
STO_WRONG_ARGUMENTS	
STO_SMALL_BUFFER	

Function 0x01

`stGetInfoFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor, fpByte buffer)***

This function uses Generic buffer with layout shown in Table 26 and further specified in Table 27 to Table 30.

Table 26: Buffer of stGetInfoFn Function

Size	Name	Comment
Doubleword	Size of data in this buffer in Bytes excluding size field	Input & Output
Vary	Section	Output
Vary	Section	Output
Etc.		

NOTES:
1. Function descriptor for this function is 0x00043X01.

This function has few sub-functions.

- Next sub-functions are identified.

Library level: requested by setting 0 in device byte of Module Descriptor

0 – General information – This common information in specified in Table 27

1 – Media information – allows caller to estimate data buffer sizes. This field is shown in Table 28

2 – Protocol information. Protocol information includes the types of security protocols supported by the SP Library as well as specifies types, sizes and usage of security objects. This allows the caller to construct input buffers. Protocol information is shown in Table 29

Device level: requested by setting non-zero value in device byte of Module Descriptor. Valid range is determined by value of number of supported interfaces and devices as shown in Table 27 and Table 30

0 – Device general information. This common information in specified in Table 30

1 – Device media information – is shown in Table 31

Table 27: Buffer of stGetInfoFn Function, sub-function 0 – General Information, Library Level

Size	Name	Comment
Doubleword	Size of data in this buffer in Bytes excluding size field	Input & Output
Word	Section size	Output
Word	Section tag	Output
16 Bytes	Storage Library Name from Export Structure name field [1]	Output
Word	Minor revision of SP Library	Output
Word	Major revision of SP Library	Output
Word	Number of interfaces [2]	Output

NOTES:

1. *Name* is ASCII string with either length of 16 bytes or zero padded to the length of 16 bytes.
2. Total number of interfaces supported by the library. This is static information – it does not depend on how many devices are currently actually attached or recognized by system. This number defines valid value of interface nibble in device byte of module descriptor. Generally this value is used for configuration purposes. See below to determine the interfaces that must be enabled by the BIOS on behalf of the module. Zero value in this field means that the module is completely self-encapsulated and doesn't require BIOS assistance to access device. Generally if this value is 0 then storage configuration functions don't need to be implemented – see below.
3. Function descriptor for this function is 0x00043001.
4. This sub-function must be supported.

Table 28: Buffer of stGetInfoFn Function, Sub-function 1 – Media Information, Library Level

Size	Name	Comment
Doubleword	Size of data in this buffer in Bytes excluding size field	Input & Output
Word	Section size	Output
Word	Section tag	Output
Doubleword	Storage medium size in bytes [1]	Output
Doubleword	Storage slot count [2]	Output
Doubleword	Storage slot size in bytes [3]	Output
Word	Storage lock mask [4]	Output
Word	Storage unlock mask [4]	Output

NOTES:

1. Aggregate size of storage of all storage devices including hidden areas such as used for storing of security objects.
2. Aggregate slot count in all storage devices.
3. This field must be filled as follows:
   – if there is only one storage device or all devices have the same slot size, this field must contain this slot size;
   – if there are many storage devices and they have different slot sizes, this field must be 0 and this function must be supported at device level as shown in Table 30.
4. Bit mask that shows supported storage and slots lock/unlock capabilities.
5. Function descriptor for this function is 0x00043101.
6. This sub-function must be supported.

Table 29: Buffer of stGetInfoFn Function, Sub-function 2 – Protocol Information, Library Level

Size	Name	Comment
Doubleword	Size of data in this buffer in Bytes excluding size field	Input & Output
Word	Section size	Output
Word	Section tag	Output
Word	Type of supported Protocol[1]	Output
Word	Type of supported hash algorithm [2]	Output
Word	Type of supported symmetric encryption algorithm [3]	Output
Word	Length of symmetric encryption key in bits[10]	Output
Word	Number of supported Access Control Objects (ACO) per slot [4]	Output
Word	Type of Access Control Object [5]	Output
Word	Size of Access Control Object in bytes	Output
Word	Type of Protection Challenge Object [6]	Output
Word	Size of Protection Challenge Object in bytes	Output
Word	Type of Security Trailer [7]	Output
Word	Size of Security Trailer in bytes	Output
Etc.	For all supported protocols	Output

NOTES:

1. Refer to Table 12
2. Refer to Table 15
3. Refer to Table 16
4. The returned value must include the number of ACOs associated with storage and slots. For instance, if 2 ACOs are associated with the slot, this field must contain 3.
5. Refer to Table 17
6. Refer to Table 19
7. Refer to Table 20
8. Function descriptor for this function is 0x00043201.
9. This sub-function is mandatory.
10. This field may be ingnored if the symmetric encryption algorithm used implies the key length. Such as the implied 56-bit key length of DES.

Table 30: Buffer of stGetInfoFn Function, Sub-function 0 – General Information, Device Level

Size	Name	Comment
Doubleword	Size of data in this buffer in Bytes excluding size field	Input & Output
Word	Section size	Input
Word	Section tag	Input
16 bytes	Device name [1]	Output
Word	Minor revision of device	Output
Word	Major revision of device	Output
Word	Number of devices [2]	Output

NOTES:

1. Name is an ASCII string with either length of 16 bytes or zero padded to the length of 16 bytes.
2. This is dynamic information. It is available only after the device's interface controler is configured. In most cases the number of devices will be 1 and a value larger than this will be considered an error.
3. Function descriptor for this function is 0x00043001.
4. This sub-function is optional. It must be implemented if storage supports more than one device on either of interfaces or it supports multiple interfaces and device characteristics on diferent interfaces are ddifferent.

Table 31: Buffer of stGetInfoFn Function, Sub-function 1 – Media Information, Device Level

Size	Name	Comment
Doubleword	Size of data in this buffer in Bytes excluding size field	Input & Output
Word	Section size	Input
Word	Section tag	Input
Doubleword	Storage medium size in bytes [1]	Output
Doubleword	Storage slot count [2]	Output
Doubleword	Storage slot size in bytes [3]	Output
Word	Storage lock mask [4]	Output
Word	Storage unlock mask [4]	Output

NOTES:

1. Medium size of specified storage device including hidden areas such as used for storing of security objects.
2. Slot count in specified storage device.
3. Slot size in specified storage device.
4. Bit mask that shows supported storage and the slots lock/unlock capabilities.
5. Function descriptor for this function is 0x00043101.
6. This sub-function is optional. It must be implemented if storage supports more than one device on either of interfaces or if it supports multiple interfaces and the device characteristics on each interface is different.

Description of function

`stGetInfoFn` returns general purpose information about the storage SPL and individual media devices. Generally, there are two types of information returned by this function – static and dynamic. Static information constitutes hard-coded information and does not depend on storage and device initialization. Dynamic information conversely becomes available only after initialization and configuration of library and devices. It must be guaranteed that sub-function 0 of this function can be called before initialization at library level as shown in Table 27. All other forms of this function have to be called after storage initialization and configuration. This function is required at library level.

Arguments

Device byte of Module Descriptor must be set to 0 to call function at library level or to device number to call function at device level. Device byte is constructed as described in Section 4.4.4

Buffer Fields

- Input values:
  - Size field must be set to the length of allocated buffer less 4 (length of size field itself).
- Output values:
  - Buffer is filled with storage information specified in Table 27 to Table 31
  - Size field is set to the length of actually allocated data.

Error Codes

STO_FUNCTION_NOT_SUPPORTED	Can be returned only at device level.
STO_NOT_INITIALIZED	Cannot be returned if sub-function 0 is called at library level.
STO_WRONG_ARGUMENTS	
STO_WRONG_DEVICE	
STO_SMALL_BUFFER	

Function 0x02

`stGetConfigFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor, fpByte buffer)***

This function uses Generic buffer with layout shown in Table 32 and further specified in Table 33.

Table 32: Buffer of stGetConfigFn Function. Generic Layout

Size	Name	Comment
Doubleword	Size of data in this buffer in Bytes excluding size field	Input & Output
Vary	Configuration Data Section	Output
Vary	Configuration Data Section	Output
Etc.		

NOTES:
1. Function descriptor for this function is 0x00043002.

Table 33: Buffer of stGetConfigFn Function. Section Format

Size	Name	Comment
Word	Size of data in section excluding size and tag fields	Output
Word	Section tag	Output
Doubleword	Host product ID [1]	Output
Byte	Revision ID	Output
3 bytes	Host type code [1]	Output
Word	Attributes [1]	Output
Vary	Allocated resources substructure [1, 2]	Output
Vary	Possible resources substructure [1, 3]	Output

NOTES:
1. Field follows format and requirements of PNP ISA specification – see Plug and Play ISA Specification , Section 6.2
2. Placeholder for allocated resources set by stSetConfigFn function.
3. Possible resources that host needs to provide.

Description of Function

`stGetConfigFn` returns configuration information about Storage SPL and individual media devices. This information is used to convey to BIOS resource requirements for enabling of Storage. Function may be supported at library level if resource requirements are the same for every supported media device. Otherwise function must be implemented at device level. This function is optional. It needs to be implemented only if BIOS assistance in configuration is required.

Configuration data in return buffer are divided into sections as shown in Table 32. Format of section is shown in Table 33. Generally it follows PnP format as defined in PnP BIOS and PnP ISA specifications with next updates.

- Configuration requirements are related to device host – i.e. controller of bus to which device is attached.
- It is expected that BIOS enables host (powers it on) and configures it. BIOS is free to disable host after end of execution.

- Allocated resources structure is placeholder for stSetConfigFn function – its content must be 0.
- More than one configuration section may be present for Storage SPL that handles devices with different attachment. In this case all hosts have to be enabled. If Storage SPL supports alternative storages, this function must be implemented at device level
- Empty configuration section with 0 data length is valid section. Such section means that device doesn't require BIOS assistance hence, it is suggested to use more convenient way to indicate it – set "Number of interfaces" to 0 –- see Table 27 and not implement this function at all.
- Section with header and without allocated and possible resources is valid section. Such section means that BIOS is only required to enable host but not to configure it. In this situation Storage SPL is responsible for configuring of host using standard BIOS interfaces if such configuration is necessary or Storage SPL can autodetect and use any current configuration.

Arguments

Device byte of Module Descriptor must be set to 0 to call function at library level or to device number to call function at device level. Device byte is constructed as described in Section 4.4.4.

Buffer Fields

- Input values:
  - Size field must be set to the length of allocated buffer less 4 (length of size field itself).
- Output values:
  - Buffer is filled with configuration information as specified in Table 32 and Table 33
  - Size field is set to the length of actually allocated data.

Error Codes

STO_FUNCTION_NOT_SUPPORTED	Can be returned only at device level
STO_NOT_INITIALIZED	
STO_WRONG_ARGUMENTS	
STO_WRONG_DEVICE	
STO_SMALL_BUFFER	

5.5 Initialization Functions

The initialization functions are responsible for putting the storage hardware into a working state and disabling it when it is no longer needed. Content of configuration data is storage hardware specific.

Function 0x10

`stInitializeFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor, fpByte buffer)***

Buffer of this function is shown in Table 34.

Table 34: Buffer for stInitializeFn Function

Size	Field Name	Comment
Doubleword	Size of data in this buffer in Bytes excluding size field	Input
Vary	Section	Input & output
Vary	Section	Input & output
Etc.		

NOTES:
1. Function descriptor for this function 0x00043X10.
2. This function must be supported.

This function has few sub-functions.

- Next sub-functions are identified.

  Library level: requested by setting 0 in device byte of Module Descriptor

  0 – Library initialize

  1 – Reserved – shall not be used

  2 – Protocol initialize

  Device level: requested by setting non-zero value in device byte of Module Descriptor. Valid range is determined by value of number of supported devices as shown in Table 27 and Table 30.

  0 – Device initialize

Table 35: Buffer for stInitializeFn Function, sub-function 0 – Library or device initialize, all levels

Size	Field Name	Comment
Doubleword	Size of data in this buffer in Bytes excluding size field	Input
Vary	Section [1]	Input & output
Vary	Section [1]	Input & output
Etc.		

NOTES:
1. Configuration data are storage library specific and can be absent alltogether. In this case the size field must be set to 0
2. Function descriptor for this function 0x00043010.
3. This sub-function must be supported at the library level and is optional at device level.

Table 36: Buffer for stInitializeFn Function, sub-function 2 – Pprotocol initialize.

Size	Field Name	Comment
Doubleword	Size of data in this buffer in Bytes excluding size field	Input
Word	Type of supported protocol [1]	Output

NOTES:

1. Type corresponds to one of the type values returned by stGetInfoFn function – see Table 29.Function descriptor for this function 0x00043110.
2. This sub-function is optional. It needs to be supported only if Storage SPL supports more than one protocol.

Description of Function

`stInitializeFn` initializes Storage SPL or individual storage device. This function is mandatory at library level and may be implemented at device level if any of storage devices requires special form of initialization. Storage SPL is free to update content of information in buffer when function is called at any level. This can be used by Storage designer to return additional information to caller. This capability is beyond the scope of this specification.

Arguments

Device byte of Module Descriptor must be set to 0 to call function at library level or to device number to call function at device level. Device byte is constructed as described in Section 4.4.4

Buffer Fields

- Input values:
  - Size field must be set to the length of allocated buffer less 4 (length of size field itself).
  - Configuration data depend on storage implementation.
- Output values:
  - Updated content of configuration data.

Error Codes

STO_FUNCTION_NOT_SUPPORTED	Can be returned only at device level
STO_INITIALIZATION_FAILED	
STO_WRONG_ARGUMENTS	
STO_WRONG_DEVICE	

Function 0x11

`stDeinitializeFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor)***

NOTES:
1. Function descriptor for this function is 0x00002011
2. Support of this function is optional.

Description of Function

This function reverses the action of st`InitializeFn` function and de-initializes the storage hardware. BIOS may additionally need to disable storage devices after the end of execution (or use). It doesn't use any buffers and doesn't have input and output values. This function is optional. If supported this function must be the last call to Storage SPL.

Arguments

Device byte of Module Descriptor must be set to 0 to call function at library level or to device number to call function at device level. Device byte is constructed as described in Section 4.4.4.

Error Codes

STO_FUNCTION_NOT_SUPPORTED	Can be returned only at device level.
STO_NOT_INITIALIZED	
STO_WRONG_ARGUMENTS	
STO_WRONG_DEVICE	

Function 0x12

`stSetConfigFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor, fpByte buffer)***

This function is coupled with stGetConfigFn function. It uses the same buffer shown in Table 37. Detailed layout of section is shown in Table 38.

Table 37: Buffer of stSetConfigFn Function. Generic Layout

Size	Name	Comment
Doubleword	Size of data in this buffer in Bytes excluding size field	Input
Vary	Configuration Data Section	Input
Vary	Configuration Data Section	Input
Etc.		

NOTES:
1. Function descriptor for this function is 0x00043012.

Table 38: Buffer of stSetConfigFn Function. Section Format

Size	Name	Comment
Word	Size of data in section excluding size and tag fields	Output
Word	Section tag	Input
Doubleword	Host product ID [1]	Input
Byte	Revision ID	Input
3 bytes	Host type code [1]	Input
Word	Attributes	Input
Vary	Allocated resources substructure[1, 2]	Input
Vary	Possible resources substructure [1, 3]	Input

NOTES:
1. Field follow format and requirements of PNP ISA specification – see Plug and Play ISA Specification, Section 6.2.
2. Allocated resources set by stSetConfigFn function.
3. Possible resources that host may provide. This substructure is optional buf if present must correspond the same substructure returned by stGetConfigFn function.

Description of Function

`stSetConfigFn` conveys configuration information to Storage SPL and individual media devices. Function may be supported at library level if resource requirements are the same for every supported media device. Otherwise function must be implemented at device level. This function is optional. It needs to be implemented only if Storage SPL supports multiple configurations and doesn't have auto detection capabilities. This function can be unsupported even if stGetConfigFn function is implemented.

Function uses buffer shown in Table 38 . All fields are mandatory except "Possible resources substructure". Layout of configuration information in buffer must strictly follow layout returned by stGetConfigFn. Recommended way to prepare this layout is to copy selected "Possible resources substructure" fields into "Allocated resources substructure" and then update selected values. Generally this sequence follows PnP functionality with next updates.

- If this function is implemented it must be called even if return value of stGetConfigFn function indicates no BIOS assistance.

- If configuration buffer returned from stGetConfigFn contains empty section or doesn't contain "Allocated resources" substructure – the function must be called with configuration buffer unmodified.
- If stGetConfigFn returns multiple configuration sections – BIOS must configure all specified interfaces and then call this function to set all of them.

Arguments

Device byte of Module Descriptor must be set to 0 to call function at library level or to device number to call function at device level. Device byte is constructed as described in Section 4.4.4

Buffer Fields

- Input values:
  - Size field must be set to the length of allocated buffer less 4 (length of size field itself).
- Output values:
  - None

Error codes

STO_FUNCTION_NOT_SUPPORTED	Can be returned only at device level.
STO_NOT_INITIALIZED	
STO_WRONG_ARGUMENTS	
STO_WRONG_DEVICE	

5.6 Enumeration Functions

All enumeration functions are used to search storage and sequentially return information about slots that match the given criteria. These functions also establish a link between the caller and storage in the way that they:

- Initiate an anti-replay object to be generated and returned by storage and then used for validation of subsequent calls.
- Return a handle that must be used for all subsequent data access calls.

Enumeration functions use a single Header Buffer specified in Table 39.

Enumeration functions are not protected and can be called without restrictions by any caller.

Enumeration functions are mandatory at library level and can be implemented at device level.

Function 0x20

stGetFirstFn ***(DWORD moduleDescriptor, DWORD functionDescriptor, fpByte buffer)***

Table 39: Buffer of stGetFirstFn Function

Structure Size	Structure Name	Field Size	Field Name	Comments
20 bytes	Name Header	16 Bytes	Slot Name	Output
		Doubleword	Slot Descriptor	Output
Vary	Security Header	Vary	Vary	Output/Input
12	Parameter	Doubleword	Handle	Output
		Doubleword	Offset [1]	Not used
		Doubleword	Size [1]	Not used

NOTES:
1. Irrelevant, preserved
2. Function descriptor for this function 0x00043020.

Description of function

Function searches Storage and returns parameters of the first allocated slot. Free slots are ignored by this function.

Arguments

Device byte of Module Descriptor must be set to 0 to call function at library level or to device number to call function at device level. Device byte is constructed as described in Section 4.4.4

Buffer fields

- Input values:
  - Access Index.
- Output values:
  - Slot Name of the first enumerated slot
  - Slot Descriptor of the first enumerated slot
  - Handle to be used for this slot for data access

Error codes

STO_FUNCTION_NOT_SUPPORTED	Can be returned only if called at device level.
STO_NOT_INITIALIZED	
STO_WRONG_ARGUMENTS	
STO_WRONG_DEVICE	
STO_SLOT_NOT_FOUND	

Function 0x21

`stGetNextFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor, fpByte buffer)***

Table 40: Buffer of stGetNextFn Function

Structure Size	Structure Name	Field Size	Field Name	Comments
20 bytes	Name Header	16 Bytes	Slot Name	Output
		Doubleword	Slot Descriptor	Output
Vary	Security Header	Vary	Vary	Output/Input
12	Parameter	Doubleword	Handle	Input & Output
		Doubleword	Offset [1]	Not used
		Doubleword	Size [1]	Not used

NOTES:
1. Irrelevant, preserved
2. Function descriptor for this function 0x00043021.

Description of function

This function continues a search of storage started by the `stGetFirstFn`, or previous call to the `stGetNextFn` itself, and returns parameters of the next allocated slot. Free slots are ignored by this function. Both `stGetFirstFn` and `stGetNextFn` functions are used for enumeration of all used slots in storage.

Arguments

Device byte of Module Descriptor must be set to 0 to call function at library level or to device number to call function at device level. Device byte is constructed as described in Section 4.4.4

Buffer fields

- Input values:
  - Handle. It must be set to the value returned from `stGetFirstFn` function or previous call to `stGetNextFn`
  - Access Credential Index.
- Output values:
  - Slot Name of the first enumerated slot.
  - Slot Descriptor of the first enumerated slot .
  - Handle to be used for this slot for data access.

Error codes

STO_FUNCTION_NOT_SUPPORTED	Can be returned only if called at device level.
STO_NOT_INITIALIZED	
STO_WRONG_ARGUMENTS	
STO_WRONG_DEVICE	
STO_SLOT_NOT_FOUND	
STO_WRONG_HANDLE	

Function 0x22

`stGetFirstNameFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor, fpByte buffer)***

Table 41: Buffer of stGetFirstNameFn Function

Structure Size	Structure Name	Field Size	Field Name	Comments
20 bytes	Name Header	16 Bytes	Slot Name	Input
		Doubleword	Slot Descriptor	Output
Vary	Security Header	Vary	Vary	Output/Input
12	Parameter	Doubleword	Handle	Output
		Doubleword	Offset [1]	Not used
		Doubleword	Size [1]	Not used

NOTES:
1. Irrelevant, preserved
2. Function descriptor for this function 0x00043022.

Description of function

Function searches Storage and returns parameters of first allocated with specified name.

Arguments

Device byte of Module Descriptor must be set to 0 to call function at library level or to device number to call function at device level. Device byte is constructed as described in Section 4.4.4

Buffer fields

- Input values:
  - Zero padded Name of slot to be found.
  - Access Index.
- Output values:
  - Slot Descriptor of the next enumerated slot
  - Handle to be used for this slot for data access
  -

Error codes

STO_FUNCTION_NOT_SUPPORTED	Can be returned only if called at device level.
STO_NOT_INITIALIZED	
STO_WRONG_ARGUMENTS	
STO_WRONG_DEVICE	
STO_SLOT_NOT_FOUND	

Function 0x23

`stGetNextNameFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor, fpByte buffer)***

Table 42: Buffer of stGetNextNameFn Function

Structure Size	Structure Name	Field Size	Field Name	Comments
20 bytes	Name Header	16 Bytes	Slot Name	Input
		Doubleword	Slot Descriptor	Output
Vary	Security Header	Vary	Vary	Output/Input
12	Parameter	Doubleword	Handle	Input & Output
		Doubleword	Offset [1]	Not used
		Doubleword	Size [1]	Not used

NOTES:
1. Irrelevant, preserved
2. Function descriptor for this function 0x00043023.

Description of function

Function continues search of storage started by `stGetFirstNameFn`, or previous call to the `stGetNextNameFn` itself, and returns parameters of the next allocated slot with specified name. Both `stGetFirstNameFn` and `stGetNextNameFn` functions allow enumeration of all slots in storage allocated to a user/app with a specific name.

Arguments

Device byte of Module Descriptor must be set to 0 to call function at library level or to device number to call function at device level. Device byte is constructed as described in Section 4.4.4

Buffer fields.

- Input values:
  - Zero padded Name of slot to be found.
  - Handle. It must be set to the value returned from `stGetFirstNameFn` function or previous call to `stGetNextNameFn`
  - Access Index.
- Output values:
  - Slot Descriptor of the next enumerated slot
  - Handle to be used for this slot for data access

Error codes

STO_FUNCTION_NOT_SUPPORTED	Can be returned only if called at device level.
STO_NOT_INITIALIZED	
STO_WRONG_ARGUMENTS	
STO_WRONG_DEVICE	
STO_SLOT_NOT_FOUND	
STO_WRONG_HANDLE	

Function 0x24

`stGetDescrFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor, fpByte buffer)***

Table 43: Buffer of stGetDescrFn Function

Structure Size	Structure Name	Field Size	Field Name	Comments
20 bytes	Name Header	16 Bytes	Slot Name	Output
		Doubleword	Slot Descriptor	Input
Vary	Security Header	Vary	Vary	Output/Input
12	Parameter	Doubleword	Handle	Output
		Doubleword	Offset [1]	Not used
		Doubleword	Size [1]	Not used

NOTES:
1. Irrelevant, preserved
2. Function descriptor for this function 0x00043024.

Description of function

Function searches Storage for Slot using its unique Descriptor value.

Arguments

Device byte of Module Descriptor must be set to 0 to call function at library level or to device number to call function at device level. Device byte is constructed as described in Section 4.4.4

Buffer fields

- Input values:
  - Descriptor of slot to be found.
  - Access Index.
- Output values:
  - Slot Name associated with Descriptor.
  - Handle to be used for this slot for data access

Error codes

STO_FUNCTION_NOT_SUPPORTED	Can be returned only if called at device level.
STO_NOT_INITIALIZED	
STO_WRONG_ARGUMENTS	
STO_WRONG_DEVICE	
STO_SLOT_NOT_FOUND	

5.7 Allocation Functions

This set of functions includes two functions – stAllocateFn and stFreeFn. These functions are used to allocate new slots with requested parameters and remove already allocated slots.

Allocation functions are protected and require sufficient Access Rights for successful completion.

Allocation functions are mandatory at library level and can be implemented at device level.

Function 0x30

`stAllocateFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor, fpByte buffer)***

Table 44: Header Buffer of stAllocateFn Function

Structure Size	Structure Name	Field Size	Field Name	Comments
20 bytes	Name Header	16 Bytes	Slot Name	Input
		Doubleword	Slot Descriptor	Input
Vary	Security Header	Vary	Vary	Input
12	Parameter	Doubleword	Handle	Input
		Doubleword	Offset [2]	Not used
		Doubleword	Size [3]	Input

Table 45: Data Buffer of stAllocateFn Function

Field Size	Field Name			Comments
20 bytes	Name Header	16 Bytes	Slot Name to allocate	Input
		Doubleword	Slot Descriptor to allocate	Input
Vary	Security Trailer (ST)			Input

NOTES:
1. Function validates integrity of Name, Descriptor and Handle fields
2. Irrelevant, preserved
3. Size must be 20.
4. Function descriptor for this function 0x000C4030.

Description of function

This function allocates a new slot with specified in Data Buffer Name and Descriptor. If the Descriptor provided is not unique an error is returned. The Access Control Object must have asserted either "Allocate/free" AR (bit 2 is 0 – see Table 11) or "Administrate allocate/free" AR (bit 18 is 0 – see Table 11). If function is called with non-zero Access Credential Index, values of Slot Name, Slot Descriptor and Handle in Header Buffer are validated and if mismatch is found – an error is returned. Handle must be obtained from previous call to one of Enumeration functions. If function is called with zero Access Credential index all above values are irrelevant and preserved.

After a slot is allocated by this function, all Access Control Objects associated with slot are in an "initial" state. That is, they are all zeros – see note in Section 5.1. This means that if, for instance,

the Access Control Object is Pass-phrase, this initial state corresponds to intuitive notion of "No Pass-phrase". For some algorithms of computation of Security Trailers such pass-phrase will not invest value into resultant Integrity Control Value. For Access Rights mask this initial state will depend on Access Rights of allocation Requestor and generally will be Equal or Below its AR – see Section 5.2 and Table 11. This also corresponds to intuitive notion of Access Rights after slot allocation. Access Control Objects remain in this initial state until set by calling the appropriate administration functions.

The Allocation function uses both Header and Data Buffers (specified in Table 44 and Table 45).

Arguments

Device byte of Module Descriptor must be set to 0 to call function at library level or to device number to call function at device level. Device byte is constructed as described in Section 4.4.4

Buffer fields

- Input values:
  - Name of slot associated with caller. Can be blank if caller is associated with storage-wide Access Control Object (Access Credential index equal 0).
  - Descriptor of slot associated with caller. Can be blank if caller is associated with storage-wide Access Control Object (Access Credential index equal 0).
  - Protection Challenge Object may be used if required by the security protocol on the platform
  - Access Credential Index is used if required by the security protocol and must be within supported range – see section 5.2
  - Handle of slot associated with caller. Can be blank if caller is associated with storage-wide Access Control Object (Access Credential index equal 0), otherwise this value must be obtained from previous call to Enumeration function. Slot Name, Descriptor and Handle are validated by function and in case of mismatch an error is returned.
  - Size must be equal to size of Name Header (20)
  - Name of slot to allocate
  - Descriptor of slot to allocate
- Output values:
  - None

Error codes

STO_FUNCTION_NOT_SUPPORTED	Can be returned only if called at device level.
STO_NOT_INITIALIZED	
STO_WRONG_ARGUMENTS	
STO_WRONG_DEVICE	
STO_DESC_NOT_UNIQUE	
STO_NO_FREE_SLOTS	
STO_ACCESS_DENIED	Only if security protocol is supported
STO_READ_FAILED	
STO_WRITE_FAILED	
STO_ERASE_FAILED	

Function 0x31

`stFreeFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor, fpByte hBuffer, fpByte dBbuffer)***

Table 46: Header Buffer for stFreeFn Function

Structure Size	Structure Name	Field Size	Field Name	Comments
20 bytes	Name Header	16 Bytes	Slot Name [1]	Input
		Doubleword	Slot Descriptor [1]	Input
Vary	Security Header [4]	Vary	Vary	Input
12	Parameter	Doubleword	Handle [1]	Input
		Doubleword	Offset [2]	Not used
		Doubleword	Size [3]	Input

Table 47: Data Buffer for stFreeFn Function

Field Size	Field Name			Comments
20 bytes	Name Header	16 Bytes	Slot Name	Input
		Doubleword	Slot Descriptor	Input
Vary	Security Trailer (ST)			Input

NOTES:

1. Function validates integrity of Name, Descriptor and Handle fields
2. Irrelevant, preserved
3. Size must be 20.
4. For “None” protocol with Data Encryption structure of Security Header is shown in Table 14
5. Function descriptor for this function 0x000C4031.

Description of function

This function de-allocates the specified slot. The Access Control Object must have asserted either “Allocate/free” AR (bit 2 is 0 – see Table 11) or “Administrate allocate/free” AR (bit 18 is 0 – see Table 11).

Arguments

Device byte of Module Descriptor must be set to 0 to call function at library level or to device number to call function at device level. Device byte is constructed as described in Section 4.4.4

Buffer fields

- Input values:
  - Name of slot associated with caller. Can be blank if caller is associated with storage-wide Access Control Object (Access Credential index equal 0).
  - Descriptor of slot associated with caller . Can be blank if caller is associated with storage-wide Access Control Object (Access Credential index equal 0).
  - Protection Challenge Object may be used if required by the security protocol on the platform

- Access Credential Index is used if required by the security protocol and must be within supported range – see section 5.2
- Handle of slot associated with caller. Can be blank if caller is associated with storage-wide Access Control Object (Access Credential index equal 0), otherwise this value must be obtained from previous call to Enumeration or Allocation function. Slot Name, Descriptor and Handle are validated by function and in case of mismatch an error is returned.
- Size must be equal to size of Name Header (20)
- Name of slot to de-allocate.
- Descriptor of slot to de-allocate.

- Output values:
  - None

Error codes

STO_FUNCTION_NOT_SUPPORTED	Possible only when called at device level
STO_NOT_INITIALIZED	
STO_WRONG_ARGUMENTS	
STO_WRONG_DEVICE	
STO_WRONG_NAME	
STO_WRONG_DESC	
STO_WRONG_HANDLE	
STO_INSUFFICIENT_RIGHTS	
STO_ACCESS_DENIED	Only if security protocol is supported
STO_READ_FAILED	
STO_WRITE_FAILED	
STO_ERASE_FAILED	

5.8 Administration Functions

Administration functions are used to administer slots and storage. Administration functions use both Header and Data Buffers, specified in Table 23and Table 24, as parameters. All these functions use a link between the caller and storage, which must have been established prior to the function call (such as the link provided by functions from the Enumeration or Allocation groups).

Administration functions are protected and can be called only by authorized callers.

Administration functions need to be implemented if it is required by protocol. Functions can be implemented at library or device level.

Function 0x40

`stSetAccessFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor, fpByte hBuffer, fpByte dBbuffer)***

Table 48: Header Buffer of the stSetAccessFn Function

Structure Size	Structure Name	Field Size	Field Name	Comments
20 bytes	Name Header	16 Bytes	Slot Name [1]	Input
		Doubleword	Slot Descriptor [1]	Input
Vary	Security Header	Vary	Vary	Input & Output
12	Parameter	Doubleword	Handle [1]	Input & Output
		Doubleword	Offset [2]	Not used
		Doubleword	Size [3]	Input

Table 49: Data Buffer of the stSetAccessFn Function

Field Size	Field Name	Comments
Doubleword	Modify Access Control (AC) Index	Input
Byte	ACO Encryption Flag[6]	Input
Vary	New AC value	Input
Vary	Security Trailer (ST)	Input

NOTES:

1. Function validates integrity of Name, Descriptor and Handle fields. If Modify AC index is 0 (Administrative AC) the values of these fields are irrelevant and preserved.
2. Irrelevant, preserved
3. Function descriptor for this function 0x000C4040.
4. A non-zero value here indicates that the NEW ACO value is encrypted with the algorithm specified in GetInfoFn.

Description of function

This function sets Access Control Object associated with slot or with storage. Access Control Objects are indexed by their position starting from 0 up to a maximum of N-1. AC#0 is associated

with storage and remaining ACs are associated with individual slots. If the AC Index (set in the header buffer) equals the Modified AC Index (set in the data buffer) the function is called to modify Access Control Object itself. For this call to succeed the AC must have asserted the "Set AC" Access Right (bit 8 is 0–see Table 11). If Access Index differs from Modify AC Index the function is called to administer another AC. For this call to succeed the AC "Administrate" Access Right must be asserted (bit 24 is 0– see Table 11).

Arguments

Device byte of Module Descriptor must be set to 0 to call function at library level or to device number to call function at device level. Device byte is constructed as described in section 4.4.4

Buffer fields

- Input values:
  - Name of slot to modify.
  - Descriptor of slot to modify.
  - Handle. This value must be obtained from previous call to Enumeration function. Slot Name, Descriptor and Handle are validated by function and in case of mismatch an error is returned.
  - Access Index used to query access to slot.
  - Size value must be equal 4 + Size of Access Control Object – total size of data.
  - Index of Access Control Object to be set;
  - New value of Access Control Object;
- Output values:
  - Handle to be used for this slot for next data or administration access.

Error codes

STO_FUNCTION_NOT_SUPPORTED	Can be returned only if called at device level.
STO_NOT_INITIALIZED	
STO_WRONG_ARGUMENTS	
STO_WRONG_DEVICE	
STO_WRONG_NAME	
STO_WRONG_DESC	
STO_WRONG_HANDLE	
STO_INSUFFICIENT_RIGHTS	
STO_ACCESS_DENIED	
STO_READ_FAILED	
STO_WRITE_FAILED	
STO_ERASE_FAILED	

Function 0x41

`stSetRightsFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor, fpByte hBuffer, fpByte dBbuffer)***

Table 50: Header Buffer for stSetRightsFn Function

Structure Size	Structure Name	Field Size	Field Name	Comments
20 bytes	Name Header	16 Bytes	Slot Name [1]	Input
		Doubleword	Slot Descriptor [1]	Input
Vary	Security Header	Vary	Vary	Input
12	Parameter	Doubleword	Handle [1]	Input & Output
		Doubleword	Offset [2]	Not used
		Doubleword	Size [3]	Input

Table 51: Data Buffer for stSetRightsFn Function

Field Size	Field Name	Comments
Doubleword	Modify AC Index	Input
Doubleword	New AR value	Input & output
Vary	Security Trailer (ST)	Input

NOTES:

1. Function validates integrity of Name, Descriptor and Handle fields. If AC Index in Security Header is 0 (Administrative AC) the values of these fields are irrelevant and preserved.
2. Irrelevant, preserved
3. Size must be 8.
4. Function descriptor for this function 0x000C4041.

Description of function

This function sets new rights for the specified Access Control Object Index. AC#0 is associated with storage and remaining ACs are associated with individual slots. If the Access Control Object Index (set in the header buffer) equals the Modified AC Index (set in the data buffer) the function is called to modify the AR of Access Control Object itself. For this call to succeed the Access Control Object must have asserted the "Modify AR" Access Right (bit 9 must be 0– see Table 11). If Access Control Object Index differs from Modify AC Index the function is called to administer the AR of another AC. For this call to succeed the Access Control Object "Administrate" AR must be asserted (bit 25 is 0 see – Table 11).

In all cases and independently of whether function call succeeded or not, actual new AR value is returned in data buffer.

Note: Note about error reporting.

- When new slot is allocated – all access rights are asserted by default. Further access rights can be only reduced by a call to `stSetRightsFn`. When such call is received, the current AR mask is compared to requested AR mask. If reduction of any AR is requested and specific "Modify AR" is not set as described above – error is generated. If new AR is the same as old one or request is made to assert AR – such request is considered valid and no errors are returned.

This entitles the caller of this function to issue a call with a zero AR mask. Such a call will not generate errors and the net effect is that the current AR mask is returned. This is a legal way to get the current AR for the slot and storage.

- If a function is called with an invalid Access Index in the Security Header – all AR associated with this index are considered to be de-asserted which determines the function return value. No special error is generated in this case.

Arguments

Device byte of Module Descriptor must be set to 0 to call function at library level or to device number to call function at device level. Device byte is constructed as described in section 4.4.4

Buffer fields

- Input values:
  - Name of slot .
  - Descriptor of slot .
  - Handle. This value must be obtained from previous call to Enumeration function. Slot Name, Descriptor and Handle are validated by function and in case of mismatch an error is returned.
  - Access Index used to query access to slot
  - Size value must be equal 4 – total size of data.
  - Index of Access Control Object which rights are to be set;
  - New Access Rights bitmask;
- Output values:
  - Handle to be used for this slot for next data or administration access.

Error codes

STO_FUNCTION_NOT_SUPPORTED	Can be returned only if called at device level.
STO_NOT_INITIALIZED	
STO_WRONG_ARGUMENTS	
STO_WRONG_DEVICE	
STO_WRONG_NAME	
STO_WRONG_DESC	
STO_WRONG_HANDLE	
STO_INSUFFICIENT_RIGHTS	
STO_ACCESS_DENIED	
STO_READ_FAILED	
STO_WRITE_FAILED	
STO_ERASE_FAILED	

Function 0x42

`stSetLocksFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor, fpByte hBuffer, fpByte dBbuffer)***

Table 52: Header Buffer for stSetLocksFn Function

Structure Size	Structure Name	Field Size	Field Name	Comments
20 bytes	Name Header	16 Bytes	Slot Name [1]	Input
		Doubleword	Slot Descriptor [1]	Input
Vary	Security Header [4]	Vary	Vary	Input
12	Parameter	Doubleword	Handle [1]	Input & Output
		Doubleword	Offset [2]	Not used
		Doubleword	Size [3]	Input

Table 53: Data Buffer for stSetLocksFn Function

Field Size	Field Name	Comments
Doubleword	Bit Mask of Locks to change	Input
Doubleword	Bit Mask of locks to set / unset	Input
Vary	Security Trailer (ST)	Input

NOTES:

1. Function validates integrity of the Name, Descriptor and Handle fields. If the AC Index in the Security Header is 0 (Administrative AC) the values of these fields are irrelevant and preserved.
2. Irrelevant, preserved
3. Size must be 8.
4. Function descriptor for this function 0x000C4043.

Description of function

This function controls locks of the storage device and individual slots. The function can be called with any Control Object Index from 0 (Associated with Storage Administration) to N-1 (Associated with Slot Administration). If use of Protection Challenge Objects is required by a protocol, the Caller must first call any enumeration function to obtain the initial Protection Challenge Object. For this call to succeed the Access Control Object must have the "Lock" AR asserted (bit 3 or bit 19 is 0 – see Table 11) which corresponds to requested lock operation – slot locking or storage locking. It is also assumed that "Lock" AR at every level applies to all lock , unlock and master lock operations.

Function uses Bit Mask of Locks to Change – every asserted bit in this mask allows corresponding operation (locking , unlocking and master locking)– refer to Table 21: Storage and Slots Lock/Unlock Bit Definitions. The second bit mask – of locks to set, – requests new state of lock, where value 1 requests locking of object (slot or storage) and value 0 requests unlocking. Every storage device depending on its nature and implementation specifics may support only locking of objects when unlocking is possible only as a result of Hardware reset or both locking and unlocking. Caller may investigate storage capabilities by calling function stGetInfoFn, sub-function 1 – see Table 21 and Table 28. Master lock capability, if supported, has different nature.

When set, it prevents further changes of associated lock and unlock bits. Master lock can be only unset by Storage Hardware Reset.

In all cases, regardless of whether the function call succeeded or not, the actual new lock bit mask is returned in data buffer. This new lock bit mask is returned for all locks including those masked by Bit Mask of Locks to Change.

Note: Zero Mask of Locks to Change

- When function is called with zero Mask of Locks to Change, the current state of locks will be returned to the caller. This is legal way to get current lock mask.

Note: Note about error reporting.

If function is called with invalid Access Index in Security Header – all AR associated with this index are considered to be de-asserted and this determines function return value. No special error is generated in this case.Arguments

Device byte of Module Descriptor must be set to 0 to call function at library level or to device number to call function at device level. Device byte is constructed as described in Section 4.4.4

Buffer fields

- Input values:
  - Name of slot.
  - Descriptor of slot.
  - Handle. This value must be obtained from previous call to Enumeration function. Slot Name, Descriptor and Handle are validated by function and in case of mismatch an error is returned.
  - Access Control Index used to query access.
  - Size value must be equal 8 – total size of data.
  - Bit Mask of Locks to change. Every set bit in this mask specifies that corresponding lock bit is allowed to be changed – see Table 21.
  - Bit Mask of Locks to set / unset. Every set bit in this mask requests corresponding lock type to be asserted and every unset bit requests de-assertion of lock.
- Output values:
  - None

Error codes

STO_FUNCTION_NOT_SUPPORTED	
STO_NOT_INITIALIZED	
STO_WRONG_ARGUMENTS	
STO_WRONG_DEVICE	
STO_INSUFFICIENT_RIGHTS	
STO_ACCESS_DENIED	Only if security protocol is supported
STO_LOCK_FAILED	
STO_LOCK_NOT_SUPPORTED	

5.9 Data Access Functions

Data access functions are used to read and write data into slots. Data access functions use both Header and Data Buffers, specified in Table 23 and Table 24, as parameters. All these functions use a link between the caller and storage, which must have been established prior to the function call (such as the link provided by functions from the Enumeration group).

Data access functions are protected and can be called only by authorized callers.

Data access functions are mandatory and can be implemented at library level and also at device level.

Function 0x50

`stReadFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor, fpByte hBuffer, fpByte dBuffer)***

Table 54: Header Buffer for stReadFn Function

Structure Size	Structure Name	Field Size	Field Name	Comments
20 bytes	Name Header	16 Bytes	Slot Name [1]	Input
		Doubleword	Slot Descriptor [1]	Input
Vary	Security Header	Vary	Vary	Input & Output
12	Parameter	Doubleword	Handle [1]	Input & Output
		Doubleword	Offset	Input & Output
		Doubleword	Size	Input & Output

Table 55: Data Buffer for stReadFn Function

Field Size	Field Name	Comments
Vary	Data [3]	Output
Vary	Security Trailer (ST)	Input & Output

NOTES:

1. Function validates integrity of Name, Descriptor and Handle fields.
2. Placeholder with any random data on input.
3. Function descriptor for this function 0x000C4050.

Description of function

This function reads data from a specified slot starting from offset specified in Header buffer. The offset is calculated from the beginning of slot in bytes. On output, the value of the Offset field in Header buffer is incremented by the number of bytes actually read. The Size field in Header buffer specifies the size of data to be read in bytes. On output, this value is decremented by the number of bytes actually read.

Arguments

Device byte of Module Descriptor must be set to 0 to call function at library level or to device number to call function at device level. Device byte is constructed as described in Section 4.4.4

Buffer fields

- Input values:
  - Name of slot to read data from.
  - Descriptor of slot to read data from.
  - Handle. This value must be obtained from previous call to Enumeration or Data access function. Slot Name, Descriptor and Handle are validated by function and in case of mismatch an error is returned.
  - Access Index used to query access to slot
  - Size of data to read.
  - Slot offset to begin read from
- Output values:
  - Handle to be used for next access to this slot.
  - Offset incremented on value of data read
  - Size decremented on value of data read
  - Data from slot returned into Data buffer

Error codes

STO_FUNCTION_NOT_SUPPORTED	Possible only when called at device level
STO_NOT_INITIALIZED	
STO_WRONG_ARGUMENTS	
STO_WRONG_DEVICE	
STO_WRONG_NAME	
STO_WRONG_DESC	
STO_WRONG_HANDLE	
STO_INSUFFICIENT_RIGHTS	
STO_ACCESS_DENIED	Only if security protocol is supported
STO_READ_FAILED	

Function 0x51

`stWriteFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor, fpByte hBuffer, fpByte dBuffer)***

Table 56: Header Buffer for stWriteFn Function

Structure Size	Structure Name	Field Size	Field Name	Comments
20 bytes	Name Header	16 Bytes	Slot Name [1]	Input
		Doubleword	Slot Descriptor [1]	Input
Vary	Security Header	Vary	Vary	Input & Output
12	Parameter	Doubleword	Handle [1]	Input & Output
		Doubleword	Offset	Input & Output
		Doubleword	Size	Input & Output

Table 57: Data Buffer for stWriteFn Function

Field Size	Field Name	Comments
Vary	Data	Input
Vary	Security Trailer (ST)	Input & Output

NOTES:
1. Function validates integrity of Name, Descriptor and Handle fields.
2. Function descriptor for this function 0x000C4051.

Description of function

This function writes data into the specified slot starting from offset specified in Header buffer. The offset is calculated from the beginning of slot in bytes. On output, the value of the Offset field in Header buffer is incremented by the number of bytes actually written. The Size field in Header buffer specifies the size of data to be written in bytes. On output, this value is decremented by number of bytes actually written.

Arguments

Device byte of Module Descriptor must be set to 0 to call function at library level or to device number to call function at device level. Device byte is constructed as described in section 4.4.4

Buffer fields

- Input values:
  - Name of slot to write data to.
  - Descriptor of slot to write data from.
  - Handle. This value must be obtained from previous call to Enumeration or Data access function. Slot Name, Descriptor and Handle are validated by function and in case of mismatch an error is returned.
  - Access Index used to query access to slot.
  - Size of data to write.
  - Slot offset to begin writing to.

- Output values:
  - Handle to be used for next access to this slot.
  - Offset incremented on value of data written.
  - Size decremented on value of data written.

Error codes

STO_FUNCTION_NOT_SUPPORTED	Possible only when called at device level
STO_NOT_INITIALIZED	
STO_WRONG_ARGUMENTS	
STO_WRONG_DEVICE	
STO_WRONG_NAME	
STO_WRONG_DESC	
STO_WRONG_HANDLE	
STO_INSUFFICIENT_RIGHTS	
STO_ACCESS_DENIED	Only if security protocol is supported
STO_READ_FAILED	
STO_WRITE_FAILED	
STO_ERASE_FAILED	

6.0 Token Service Provider Library

This section specifies structure and interface of a Token Service Provider Library (SPL). In the following descriptions, **fpByte** designates the Far Pointer to the Byte Buffer.

6.1 Interface

All Token Interface functions can be subdivided into the following groups.

- Information functions provide information on Token SPL and module interface.
- Initialization functions allow to initialized authentication hardware.
- Log-in functions allow to establish link between library and authentication hardware (to "open" device).
- Capture functions allow to obtain User Templates.
- Verification functions allow to compare User Templates for authentication purposes.

Not all of the functions of Token SPLs interface are mandatory. Functions and function categories implemented in any specific Token depend on Token technologies and other factors. Possible implementation scenarios will be considered in later sections. Description of each optional function also contains directions to other functions, which it can be combined with.

6.1.1 Template structure

The common template structure shown below is used as a parameter in many of Token SPL functions. For now, it is assumed that total length of the template structure will fit into one slot.

Table 58: User Template Structure

Size	Name	Field Name	Comment
Word	Size of data in template. [1]		
Word	Template Tag [1]		
16 Bytes	User Name space padded [2]		
8 Bytes	Time and Date stamps [3]	Seconds	
		Minutes	
		Hours	
		Day of week	Can be left blank
		Day of month	
		Month	
		Year	
		Century	Can be left blank
Byte	User Privilege Level [4]		
Byte	Score [5]		
Word	Attribute [6]		
Vary	Template Data [7]		

NOTES:

1. Size of this field is excluded from total size of template.
2. The User Name is used for display purposes and as searchable tag when storage is scanned for specified User Template. If spaces and tabs are allowed as parts of User Name this will pose a problem because these invisible characters can be inserted as Trailering ones and create ambiquity. This is why after insertion of User Name all Trailering white spaces must be trancated and the User Name must be space-padded. Effectively the User Name is space terminated.
3. Time and Date stamps allow BIOS to enforce policy of timed credentials with potential to expire. All values in BCD format
4. User Priviledge Level lets BIOS to fine-tune User management policy. User Privilege Level is in scale from 0 to 100.
5. Score is used to store and return the result of match of captured and stored templates. Score is computed in scale from 0 to 100 and is intended for use primarily by biometric devices. Other devices must assume only two extrim values – 100 to indicate match and 0 to indicate mismatch.
6. Attribute is appointed to store template associated control flags.
7. Bit 0 = 1 – User template is suspended, otherwise it is active.
8. Template data are specific for AuD and can have any length and content. This is responsibility of Token SPL to interpret data.

6.2 Information Functions

Functions of this group provide information about interface and all supported by this Token SPL functions and also common information regarding this particular library.

Function 0x00

`tkGetCapabilitiesFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor, fpByte buffer)***

On return from function, the buffer must be filled with the information presented in Table 59.

Table 59: Buffer of tkGetCapabilitiesFn Function

Size	Name	Comment
Doubleword	Size of data in this buffer in Bytes excluding this size field	Input & output
Doubleword	Fully qualified Function Descriptor of the First supported function	Output
Doubleword	Fully qualified Function Descriptor of the Second supported function	Output
…	Etc. for all supported functions	Output

NOTES:
1. Function descriptor for this function is 0x00043000.

Description of function

`tkGetCapabilitiesFn` function provides information about Token SPL interface – function descriptors of all supported functions. For uniformity this function returns information about itself as well. This function is required and format and order of all parameters must be strictly followed. This function can be called before initialization of Token Library

Arguments

When called with Module Descriptor Device byte equal to 0 function returns descriptors of functions implemented at library level. When called with Module Descriptor Device byte not equal to 0 function returns descriptors of functions implemented at device level. Generally implementation of any of functions at device level is optional including `tkGetCapabilitiesFn`. If not implemented at device level the function must return error. This also means that other functions are also not supported at device level.

It is also assumed that the same set of functions is supported for all installed devices and thus output of `tkGetCapabilitiesFn` is the same for any of non-zero Device byte values.

Buffer fields

- Input values:
  - Size field must be set to the length of allocated buffer less 4 (length of size field itself).
- Output values:
  - Buffer is filled with Function Descriptors of all supported functions
  - Size field is set to the length of actually allocated data.

Error codes

TOK_FUNCTION_NOT_SUPPORTED	Can be returned only if called at device level.
TOK_WRONG_ARGUMENTS	
TOK_SMALL_BUFFER	

Function 0x01

`tkGetInfoFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor, fpByte buffer)***

Table 60: Buffer of tkGetInfoFn Function

Size	Name	Comment
Doubleword	Size of data in this buffer in Bytes excluding size field	Input & Output
Vary	Section	Output
Vary	Section	Output
Etc.		

NOTES:
1. Function descriptor for this function is 0x00043x01.

This function has few sub-functions.

- Next sub-functions are identified.

  Library level: requested by setting 0 in device byte of Module Descriptor

  0 – General information. This common information in specified in Table 61

  Device level: requested by setting non-zero value in device byte of Module Descriptor. Valid range is determined by value of number of supported interfaces and devices as shown in Table 61 and Table 62

  0 – AuD general information. This common information in specified in Table 62

Table 61: Buffer for tkGetInfoFn Function, sub-function 0 – general information, library level

Size	Name	Comment
Doubleword	Size of data in this buffer in Bytes excluding size field	Input & output
Word	Section size	Output
Word	Section tag	Output
16 Bytes	Token SPL Name from Export Structure name field [1]	Output
Word	Minor revision of SP library	Output
Word	Major revision of SP library	Output
Word	Number of interfaces [2]	Output

NOTES:
1. Name is ASCII string with either length of 16 bytes or zero padded to the length of 16 bytes.
2. Total number of interfaces supported by the library. This is static information – it does not depend on how many devices are currently actually attached or recognized by system. This number defines valid value of interface nibble in Device byte of module descriptor. Generally this value is used for configuration purposes – see below, – to determine the interfaces that must be enabled by BIOS on behalf of module. Zero value in this field means that module is completely self-encapsulated and doesn't require BIOS assistance to access device. Generally if this value is 0 then token configuration functions don't need to be implemented.
3. Function descriptor for this function is 0x00043001.

Table 62: Buffer for tkGetInfoFn Function, sub-function 0 – general information, device level

Size	Name	Comment
Doubleword	Size of data in this buffer in Bytes excluding size field	Input & output
Word	Section size	Output
Word	Section tag	Output
16 Bytes	Authentication Device Name [1]	Output
Word	AuD Minor revision	Output
Word	AuD Major revision	Output
Word	Number of devices [2]	Output

NOTES:

1. Name is ASCII string with either length of 16 bytes or zero padded to the length of 16 bytes.
2. This is dynamic information. It is available only after device's interface controler is configured. In most cases this number of devices will be 1 and more than one device attached to the same interface will be considered as error.
3. Function descriptor for this function is 0x00043001.

Description of function

`tkGetInfoFn` returns general purpose information about Token SPL and individual devices. Generally, there are two types of information returned by this function – static and dynamic. Static information constitutes hard-coded information and does not depend on library and device initialization. Dynamic information conversely becomes available only after initialization and configuration of library and devices. It must be guaranteed that sub-function 0 of this function can be called before initialization at library level as shown in Table 61. All other forms of this function have to be called after Token SP initialization and configuration. This function is required at library level.

Arguments

Device byte of Module Descriptor must be set to 0 to call function at library level or to device number to call function at device level. Device byte is constructed as described in Section 4.4.4

Buffer fields

- Input values:
  - Size field must be set to the length of allocated buffer less 4 (length of size field itself).
- Output values:
  - Buffer is filled with specified in Table 61 to Table 62 information
  - Size field is set to the length of actually allocated data.

Error codes

TOK_FUNCTION_NOT_SUPPORTED	Can be returned only at device level.
TOK _NOT_INITIALIZED	Cannot be returned if sub-function 0 is called at library level.
TOK _WRONG_ARGUMENTS	
TOK _WRONG_DEVICE	
TOK _SMALL_BUFFER	

Function 0x02

`tkGetConfigFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor, fpByte buffer)***

This function uses Generic buffer with layout shown in Table 63 and further specified in Table 64.

Table 63: Buffer of tkGetConfigFn Function. Generic layout

Size	Name	Comment
Doubleword	Size of data in this buffer in Bytes excluding size field	Input & Output
Vary	Configuration Data Section	Output
Vary	Configuration Data Section	Output
Etc.		

NOTES:
1. Function descriptor for this function is 0x00043002.

Table 64: Buffer of tkGetConfigFn Function. Section format

Size	Name	Comment
Word	Size of data in section excluding size and tag fields	Output
Word	Section tag	Output
Doubleword	Host product ID [1]	Output
Byte	Revision ID	Output
3 bytes	Host type code [1]	Output
Word	Attributes [1]	Output
Vary	Allocated resources substructure [1, 2]	Output
Vary	Possible resources substructure [1, 3]	Output

NOTES:
1. Field follow format and requirements of PNP ISA specification – see Plug and Play ISA Specification, Section 6.2
2. Placeholder for allocated resources set by stSetConfigFn function.
3. Possible resources that host needs to provide.

Description of function

`tkGetConfigFn` returns configuration information about Token SPL and individual authentication devices. This information is used to convey to BIOS resource requirements for enabling of Token SPL. Function may be supported at library level if resource requirements are the same for every supported AuD. Otherwise function must be implemented at device level. This function is optional. It needs to be implemented only if BIOS assistance in configuration is required.

Configuration data in return buffer are divided into sections as shown in Table 63. Format of section is shown in Table 64. Generally it follows PnP format as defined in PnP BIOS and PnP ISA specifications with next updates.

- Configuration requirements are related to device host – i.e. controller of bus to which device is attached.
- It is expected that BIOS enables host (powers it on) and configures it. BIOS is free to disable host after end of execution.
- Allocated resources structure is placeholder for tkSetConfigFn function – its content must be 0.

- More than one configuration section may be present for Token SPL that handles devices with different attachment. This is indicated by "Number of interfaces" field in Table 61. In this case all hosts have to be enabled.
- Empty configuration section with 0 data length is valid section. Such section means that device doesn't require BIOS assistance, hence it is suggested to use more convenient way to indicate it – set "Number of interfaces" to 0 –- see Table 61 and not implement this function at all.
- Section with header and without allocated and possible resources is valid section. Such section means that BIOS is only required to enable host but not to configure it. In this situation Token SPL is responsible for configuring of host using standard BIOS interfaces.

Arguments

Device byte of Module Descriptor must be set to 0 to call function at library level or to device number to call function at device level. Device byte is constructed as described in section 4.4.4

Buffer fields

- Input values:
  - Size field must be set to the length of allocated buffer less 4 (length of size field itself).
- Output values:
  - Buffer is filled with configuration information as specified in Table 63 and Table 64
  - Size field is set to the length of actually allocated data.

Error codes

TOK_FUNCTION_NOT_SUPPORTED	Can be returned only if called at device level.
TOK _NOT_INITIALIZED	
TOK _WRONG_ARGUMENTS	
TOK _WRONG_DEVICE	
TOK _SMALL_BUFFER	

6.3 Initialization Functions

The initialization functions are responsible for putting the authentication hardware into a working state and disabling it when it is no longer needed.

Examples of operations that can be performed by initialization functions are – detecting relevant hardware addresses, device calibration, establishing appropriate device parameters such as current, voltage, time-outs etc. The exact operations performed by a particular Token initialization depends on the type of Token.

Next are assumptions that control how particular Token Authentication Device (AuD) is initialized.

- Device must be initially enabled by BIOS. Token SPL is responsible for device initialization but is not responsible for device enabling. This enabling includes but is not limited to powering of device on, enabling of bus to which device is attached, making device visible on this bus etc.
- Token SPL can use all standard and public BIOS services to communicate with the AuD or it can completely incorporate interface to the device in its own code.
- If the AuD is configurable and the BIOS exports standard means for configuring devices of this type, the Token SPL can employ such configuration services. Otherwise, it is responsibility of the BIOS to configure the AuD and convey this configuration information to the Token SPL.

Function 0x10

`tkInitializeFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor, fpByte buffer)***

This function has few sub-functions.

- Next sub-functions are identified.

  Library level: requested by setting 0 in device byte of Module Descriptor

  0 – Library initialize

  Device level: requested by setting non-zero value in device byte of Module Descriptor. Valid range is determined by value of number of supported devices as shown in Table 61 and Table 62

  0 – Device initialize

  Buffer of this function is shown in Table 65 and Table 66.

Table 65: Buffer for tkInitializeFn Function, sub-function 0 – library initialize, library level

Size	Structure Name	Field Name	Comment
Doubleword		Size of data in this buffer in Bytes excluding size field [1]	Input & output
Word	Configuration data first section	Section Data Size	Input & output
Word		Section Tag	Input & output
Vary		Section Data	Input & output
Word	Configuration data second section	Section Data Size	Input & output
Word		Section Tag	Input & output
Vary		Section Data	Input & output
….	Etc.		

NOTES:
1. Function can neither change the size of configuration buffer nor to allocate additional configuration slots. Thus this value must be preserved.
2. Function descriptor for this function 0x00043010.
3. This function must be supported.

Table 66: Buffer for stInitializeFn Function, sub-function 0 – Device initialize, device level

Size	Field Name	Comment
Doubleword	Size of data in this buffer in Bytes excluding size field	Input
Vary	Configuration Data [1]	Input & output

NOTES:
1. Configuration data are Token AuD specific and can be absent altogether. In this case the size field must be set to 0
2. Function descriptor for this function 0x00043010.
3. This function is optional.

Description of function

At library level this function is implemented as follows. Before calling of function the caller must enumerate all configuration slots in storage that belong to Token SPL and obtain common size of

these slots. Then the caller must allocate a buffer with a size not less than the common size of all enumerated slots. The caller then sets the size field, in accordance with Table 72, to the value of the allocated buffer less 4 (length of size field itself). The caller then reads all configuration slots sequentially into the allocated buffer, thus making all of the initialization data available to the function. Initialization data are allocated in sections. All types of sections are optional but some of them are highly recommended.

Section Classifications:

1. "Friendly SP Name". This section contains the string used as Token SP name displayed in User Authentication Menus. If not present, the 16-byte Token SP name, extracted from the initialization slot is used instead. There can be only one such section in the initialization buffer. This section is recommended in all cases. Format of section is shown in Table 67

2. "Custom Error Message". These sections contain strings used to display on screen custom error messages with error codes ERROR_BASE + 0x800 and above. If not present for specific error code, caller is free to choose format of message. These sections are recommended for all custom error codes. Format of section is shown in Table 68

3. "Configuration Data". These sections contain token configuration data and are described with tkGetConfigFn and tkSetConfigFn functions.

Initialization data buffer may also contain sections with custom information specific to Token SP. Such sections must be tagged with Cutom Tags – see Appendix E. Interpretation of data in such sections is totally determined by the token designer. During execution, the function can change configuration data. So after a function returns the caller must check the integrity of the data. If content of the buffer changed – the Caller writes buffer back to configuration slots.

At device level the structure of initialization information is totally determined by Token designer.

Table 67: Format of "Friendly SP Name" section

Size	Field Name	Comment
Word	Size of data in this section in Bytes excluding size and tag field s	
Word	Section Tag	
Vary	Friendly SP name string [1]	

NOTES:
1. Not sero terminated

Table 68: Format of "Custom Error Message" section

Size	Field Name	Comment
Word	Size of data in this section in Bytes excluding size and tag field s	
Word	Section Tag	
Doubleword	Error code	
Vary	Friendly SP name string [1]	

NOTES:
1. Not sero terminated

This function is mandatory at the library level and may be implemented at the device level if any of devices requires special form of initialization. Token SPL is free to update content of information in buffer when function is called at device level. This can be used by the Token

designer to return additional information to the caller. This capability is beyond the scope of this specification.

Arguments

Device byte of Module Descriptor must be set to 0 to call function at library level or to device number to call function at device level. Device byte is constructed as described in Section 4.4.4

Buffer fields

- Input values:
  - Size field must be set to the length of allocated buffer less 4 (length of size field itself).
  - All configuration slots data sequentially read into function Buffer
- Output values:
  - Updated content of configuration slots

Error codes

TOK_FUNCTION_NOT_SUPPORTED	Can be returned only at device level
TOK _INITIALIZATION_FAILED	
TOK _WRONG_ARGUMENTS	
TOK _WRONG_DEVICE	
TOK_NO_BIOS_SERIAL_SERVICE	
TOK_NO_BIOS_PARALLEL_SERVICE	
TOK_NO_BIOS_USB_SERVICE	
TOK_NO_BIOS_1394_SERVICE	
TOK_TOKEN_NOT_FOUND	
TOK_MORE_THAN_ONE_TOKEN	
TOK_DEVICE_READ_FAILED	

Function 0x11

NOTES: `tkDeinitializeFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor)***

1. Function descriptor for this function is 0x00002011
2. Support of this function is optional.

Description of function

This function reverses the action of `tkInitializeFn` function and de-initializes the Token SPL or device. BIOS may additionally need to disable a device after the end of execution (or use). It doesn't use any buffers and doesn't have input and output values. This function is optional. If supported this function must be the last call to Token SPL.

Arguments

Device byte of Module Descriptor must be set to 0 to call function at library level or to device number to call function at device level. Device byte is constructed as described in Section 4.4.4

Error codes

TOK_FUNCTION_NOT_SUPPORTED	Can be returned only if called at device level.
TOK_NOT_INITIALIZED	
TOK _WRONG_ARGUMENTS	
TOK _WRONG_DEVICE	

Function 0x12

`tkSetConfigFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor, fpByte buffer)***

This function is coupled with tkGetConfigFn function. It uses the same buffer shown in Table 69. Detailed layout of section is shown in Table 70.

Table 69: Buffer of tkSetConfigFn Function. Generic layout

Size	Name	Comment
Doubleword	Size of data in this buffer in Bytes excluding size field	Input
Vary	Configuration Data Section	Output
Vary	Configuration Data Section	Output
Etc.		

NOTES:
1. Function descriptor for this function is 0x00043012.

Table 70: Buffer of tkSetConfigFn Function. Section format

Size	Name	Comment
Word	Size of data in section excluding size and tag fields	Output
Word	Section tag	Output
Doubleword	Host product ID [1]	Output
Byte	Reserved	Not used
3 bytes	Host type code [1]	Output
Word	Attributes [1]	Output
Vary	Allocated resources substructure[1, 2]	Output
Vary	Possible resources substructure [1, 3]	Output

NOTES:
1. Field follow format and requirements of PNP ISA specification – see Plug and Play ISA Specification , Section 6.2.
2. Allocated resources set by stSetConfigFn function.
3. Possible resources that host needs to provide.

Description of function

`tkSetConfigFn` conveys configuration information to Token SPL and individual authentication devices. Function may be supported at library level if resource requirements are the same for every supported device. Otherwise function must be implemented at device level. This function is optional. It needs to be implemented only if Token SPL supports multiple configurations and doesn't have auto-detection capabilities. This function can be unsupported even if stGetConfigFn function is implemented.

Function uses the buffer in Table 70. All fields are mandatory except "Possible resources substructure". Layout of configuration information in buffer must strictly follow layout returned by stGetConfigFn. Recommended way to prepare this layout is to copy selected "Possible resources substructure" fields into "Allocated resources substructure" and then update selected values. Generally this sequence follows PnP functionality with next updates.

- If configuration buffer returned from stGetConfigFn contains empty section or doesn't contain "Allocated resources" substructure – the function must be called with configuration buffer unmodified.

- If stGetConfigFn returns multiple configuration sections – BIOS must configure all specified interfaces and then call this function to set all of them.

Arguments

Device byte of Module Descriptor must be set to 0 to call function at library level or to device number to call function at device level. Device byte is constructed as described in Section 4.4.4

Buffer fields

- Input values:
  - Size field must be set to the length of allocated buffer less 4 (length of size field itself).
- Output values:
  - None.

Error codes

TOK_FUNCTION_NOT_SUPPORTED	Can be returned only if called at device level.
TOK _NOT_INITIALIZED	
TOK _WRONG_ARGUMENTS	
TOK _WRONG_DEVICE	

6.4 Log-in Functions

Some authentication devices must be “opened” before they can be used (such as smart cards). These devices may be PIN or PASSWORD protected. To open such a device the “unlock” command sent to device must include this log-in information. The functions of the Log-in group enable this capability. All functions in this group are optional and must be implemented only if AuD supports lock/unlock functionality.

Only one of the functions from this group actually needs to be implemented. It is the SPL vendor that decides which of these functions to implement.

Log-in functions may be implemented at library or device level.

Function 0x20

`tkLoginFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor, fpByte buffer)***

Table 71: Buffer for tkLoginFn Function

Size	Name	Comment
Doubleword	Size of data in this buffer in Bytes excluding size field [1]	Input
16 Bytes	User Personal Identification Number (PIN) [2]	Input

NOTES:
1. This value equals 16.
2. PIN must be zero padded. PIN is optional and if not present all 0s must be entered in this field.
3. Function descriptor for this function is 0x00043020
4. This function is optional.

Description of function

This function establishes connection between Token SPL and Authentication Device (AuD). If AuD is PIN protected the caller must collect user PIN, before calling this function, and insert it into the input buffer using zero padding. A PIN is optional, it can be skipped if the device is not PIN protected. To skip PIN all 0s must be entered in the PIN field of input buffer.

Arguments

Device byte of Module Descriptor must be set to 0 to call function at library level or to device number to call function at device level. Device byte is constructed as described in Section 4.4.4

Buffer fields

- Input values:
  - Size field must be set to 16.
  - Zero padded PIN or all 0s if PIN is not used.
- Output values:
  - None

Error codes

TOK_FUNCTION_NOT_SUPPORTED	Can be returned only if called at device level.
TOK _NOT_INITIALIZED	
TOK _WRONG_ARGUMENTS	
TOK _WRONG_DEVICE	
TOK_TOKEN_INVALID	
TOK_TEMPLATE_INVALID	
TOK_LOGIN_FAILED	

Function 0x21

`tkLoginExtFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor)***

NOTES:

1. The Function Descriptor for this function is 0x00002021.
2. This function is optional.

Description of function

This function is a variation of the `tkLoginFn` function described above. It can be used in next cases.

- When AuD is not PIN protected. This is equivalent to calling of `tkLoginFn` with PIN field set to 0;
- When AuD is connected to system through proprietary interface. Example of this is a PINPAD on a smartcard reader but also may be some special mechanism on the PC platform itself.
- Such interface can enable very secure algorithm of User PIN collection without exposure of it to system software. In this case `tkLoginExtFn` only initiates the process of PIN collection.

This function like `tkLoginFn` establishes connection between Token SPL and Authentication Device (AuD) and also like `tkLoginFn` is optional.

Arguments

Device byte of Module Descriptor must be set to 0 to call function at library level or to device number to call function at device level. Device byte is constructed as described in Section 4.4.4

Error codes

TOK_FUNCTION_NOT_SUPPORTED	Can be returned only if called at device level.
TOK _NOT_INITIALIZED	
TOK _WRONG_ARGUMENTS	
TOK _WRONG_DEVICE	
TOK_TOKEN_INVALID	
TOK_TEMPLATE_INVALID	
TOK_LOGIN_FAILED	

6.5 Capture Functions

The functions of this group are responsible for returning high quality data that will be later used to authenticate a user. This data is treated as "opaque" data from the BIOS perspective. That is, it will save it in protected storage as-is (except a few defined fields are added) and retrieve it as-is and let the Token SPL be responsible for interpreting it.

- For biometric devices the data may be "template" – extract of gray-scale image processed internally by Token SPL suitable for matching operation.
- For the smart card genre, the data may be a hashed secret or a certificate, etc.
- For intelligent devices (remote matching engine and storage) the data may be secret use later to validate a go/no-go response, etc.

Capture functions can be implemented at library or device level.

Due to the nature of this group of functions, one and only one (of Capture, Register or Enroll) must be implemented. This is illustrated in Table 79. If Register function is implemented then it is assumed that Capture and Match functions are implemented as well. If Enroll is implemented then Erase is also needed.

Function 0x30

`tkCaptureFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor, fpByte buffer)***

This function can be called with Module Descriptor Device Byte equal to 0 or set to sequential number of device.

Table 72: Buffer for tkCaptureFn Function

Size	Name	Comment
Word	Size of data in this buffer in Bytes excluding size and tag fields [1]	Input
Word	Tag [2]	
16 Bytes	User Name space padded [2]	Total 28 bytes
8 Bytes	Time and Date stamps [2]	
Byte	User Privilege Level [2]	
Byte	Score [2]	
Word	Attribute [2]	
Vary	Template Data	Output

NOTES:

1. This value excludes size of Tag and Size fields.
2. These fields are used by caller of this function. Function ignores these values and must not alter them.
3. Function descriptor for this function is 0x00043030.
4. Function is optional.

Description of function

This function captures information from the AuD and returns a Authentication Template suitable for storing in the platform protected storage. This same data can be retrieved later and used to perform a "match" operation.

If there is only one device attached to system or Token SPL design supports simultaneously only one device then the value of Module Descriptor Device Byte is "don't care". Otherwise the function captures data from specified device.

Arguments

Device byte of Module Descriptor must be set to 0 to call function at library level or to device number to call function at device level. Device byte is constructed as described in Section 4.4.4

Buffer fields

- Input values:
  - Size field must be set to the size of allocated buffer less 4
- Output values:
  - Template data
  - Size field must be set to the size of data in buffer that is Size = Size of Template Data + 28

Error codes

TOK_FUNCTION_NOT_SUPPORTED	Can be returned only if called at device level.
TOK _NOT_INITIALIZED	
TOK _WRONG_ARGUMENTS	
TOK _WRONG_DEVICE	
TOK_TOKEN_INVALID	
TOK_TEMPLATE_INVALID	
TOK_PIN_REQUIRED	
TOK_PIN_EXPIRED	
TOK_IMAGE_QUALITY_FAILED	
TOK_TEMPLATE_LOW_SEQURITY	
TOK_READ_FAILED	

Function 0x31

`tkRegisterFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor, fpByte buffer)***

This function can be called with Module Descriptor Device Byte equal to 0 or set to sequential number of device.

Table 73: Buffer for tkRegisterFn Function

Size	Name	Comment
Word	Size of data in this buffer in Bytes excluding size and tag fields [1]	Input
Word	Tag [2]	
16 Bytes	User Name space padded [2]	Total 28 bytes
8 Bytes	Time and Date stamps [2]	
Byte	User Privilege Level [2]	
Byte	Score [2]	
Word	Attribute [2]	
Vary	Template Data	Output

NOTES:
1. This value excludes size of Tag and Size fields.
2. These fields are used by caller of this function. Function ignores these values and must not alter them.
3. Function descriptor for this function is 0x00043031.
4. Function is optional.

Description of function

This function, similar to the tkCaptureFN above, captures information from the AuD and returns a *Authentication Template* suitable for storing in the platform protected storage. This function is optional and is designed specifically enrolling or registering a template that will be subsequently be used to verify the identity of a user. The capture function can be used for this purpose when the template is simple in nature. This function provides a mechanism for the service provider to provide a differentiated template capture process that may be more thorough, involved, interactive, etc. during the first interaction with the user (registration, enrollment, etc.). If this function is implemented then it must be used to obtain a template to be stored in protected storage. The capture function is used to obtain a template that is passed to the Match function for user authentication purposes.

If there is only one device attached to system or the Token SPL supports only one simultaneous device, then the value of Module Descriptor *Device Byte* is a “don’t care”. Otherwise the function captures data from the specified device.

Arguments

Device byte of Module Descriptor must be set to 0 to call function at library level or to device number to call function at device level. Device byte is constructed as described in Section 4.4.4

Buffer fields

- Input values:
  - Size field must be set to the size of allocated buffer less 4
- Output values:
  - Template data
  - Size field must be set to the size of data in buffer that is Size = Size of Template Data + 28

Error codes

TOK_FUNCTION_NOT_SUPPORTED	Can be returned only if called at device level.
TOK _NOT_INITIALIZED	
TOK _WRONG_ARGUMENTS	
TOK _WRONG_DEVICE	
TOK_TOKEN_INVALID	
TOK_TEMPLATE_INVALID	
TOK_PIN_REQUIRED	
TOK_PIN_EXPIRED	
TOK_IMAGE_QUALITY_FAILED	
TOK_TEMPLATE_LOW_SEQURITY	
TOK_READ_FAILED	

Function 0x32

`tkEnrollFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor, fpByte buffer)***

This function can be called with Module Descriptor Device Byte equal to 0 or set to sequential number of device.

Table 74: Buffer for tkEnrollFn Function

Size	Name	Comment
Word	Size of data in this buffer in Bytes excluding size and tag fields [1]	Input
Word	Tag [2]	
16 Bytes	User Name space padded [2]	Total 28 bytes
8 Bytes	Time and Date stamps [2]	
Byte	User Privilege Level [2]	
Byte	Score [2]	
Word	Attribute [2]	
Vary	Template Identifier	Output

NOTES:

1. This value excludes size of Tag and Size fields.
2. These fields are used by caller of this function. Function ignores these values and must not alter them.
3. Function descriptor for this function is 0x00043032.
4. Function is optional but if used all parameters are required.

Description of function

This function is similar to the `tkCaptureFn` but assumes that the AuD is a "smart" subsystem with its own internal protected storage and possibly, match engine. The `tkEnrollFn` tells the subsystem to do what it needs to do to capture, extract, and save information about a user for later "authentication/identification" analysis. An identifier, secret, etc. must be returned to the IPAA subsystem. This returned information can be used by the SPL to later validate the go/no-go response from the sub-system.

If this function is implemented then `tkEraseFn and tkValidateFn` (see later section) must be implemented as well.

Arguments

Device byte of Module Descriptor must be set to 0 to call function at library level or to device number to call function at device level. Device byte is constructed as described in Section 4.4.4

Buffer fields

- Input values:
  - Size field must be set to the size of allocated buffer less 4
- Output values:
  - Template data
  - Size field must be set to the size of data in buffer that is Size = Size of Template Identifier + 28

Error codes

TOK_FUNCTION_NOT_SUPPORTED	Can be returned only if called at device level.
TOK _NOT_INITIALIZED	
TOK _WRONG_ARGUMENTS	
TOK _WRONG_DEVICE	
TOK_TOKEN_INVALID	
TOK_TEMPLATE_INVALID	
TOK_PIN_REQUIRED	
TOK_PIN_EXPIRED	
TOK_IMAGE_QUALITY_FAILED	
TOK_TEMPLATE_LOW_SEQURITY	
TOK_READ_FAILED	
TOK_WRITE_FAILED	

Function 0x33

`tkEraseFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor, fpByte buffer)***

Table 75: Buffer for tkEraseFn Function

Size	Name	Comment
Word	Size of data in this buffer in Bytes excluding size and tag fields [1]	Input
Word	Tag [2]	
16 Bytes	User Name space padded [2]	Total 28 bytes
8 Bytes	Time and Date stamps [2]	
Byte	User Privilege Level [2]	
Byte	Score [2]	
Word	Attribute [2]	
Vary	Template Identifier	Input

NOTES:

1. This value excludes size of Tag and Size fields.
2. These fields are used by caller of this function. Function ignores these values and must not alter them.
3. Function descriptor for this function is 0x00043033.
4. Function is optional but if used all parameters are required.

Description of function

This function erases template from local storage of AuD with "smart" subsystem.

If this function is implemented then `tkEnrollFn and tkValidateFn` (see later section) must be implemented as well.

Arguments

Device byte of Module Descriptor must be set to 0 to call function at library level or to device number to call function at device level. Device byte is constructed as described in section 4.4.4

Buffer fields

- Input values:
  - Size field must be set to the size of data in buffer that is Size = Size of Template Identifier + 28
  - Template identifier to be erased
- Output values:
  - None

Error codes

TOK_FUNCTION_NOT_SUPPORTED	Can be returned only if called at device level.
TOK _NOT_INITIALIZED	
TOK _WRONG_ARGUMENTS	
TOK _WRONG_DEVICE	
TOK_TEMPLATE_NOT_FOUND	
TOK_WRITE_FAILED	

6.6 Verification Functions

The purpose of the verification group is to use all the information that has be gathered and stored to "authenticate" a user that is attempting access to a PC platform. It can do this in a step-like fashion providing more granularity and code reuse, or it can be supported as an "atomic" operation to support smart subsystems and improve code security. The following is assumed:

- If a function uses two buffers then the iBuffer (input buffer) contains a template captured from AuD and the cBuffer (compare buffer) contains template read from storage.
- Both iBuffer and cBuffer have the same Template structure described in Table 58.
- Functions performs a template match regardless of the user name, time stamps, and user privilege levels. This is the caller's responsibility to interpret these values according to system policy.
- Threshold value necessary for match is supplied to Token SPL as part of configuration data.

Verification functions can be implemented at library or device level.

Function 0x40

`tkMatchFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor, fpByte iBuffer, fpByte cBuffer)***

Table 76: iBuffer and cBuffer for tkMatchFn Function

Size	Name	Comment
Word	Size of data in this buffer in Bytes excluding size and tag fields [1]	Input
Word	Tag [2]	
16 Bytes	User Name space padded [2]	
8 Bytes	Time and Date stamps [2]	
Byte	User Privilege Level [2]	
Byte	Score	Output
Word	Attribute [2]	
Vary	Template Data	Input

NOTES:

1. This value excludes size of Tag and Size fields.
2. These fields are used by caller of this function. Function ignores these values and must not alter them.
3. Function descriptor for this function is 0x000C4040.
4. Function is optional but if used all parameters are required.

Description of function

This function is used in conjunction with `tkCaptureFn` to perform match of captured and stored templates. The function is optional and if not supported then the caller must assume that the template returned by `tkCaptureFn` must have 1:1 match to template data stored in the storage.

Arguments

Device byte of Module Descriptor must be set to 0 to call function at library level or to device number to call function at device level. Device byte is constructed as described in Section 4.4.4

Buffer fields

- Input values:
  - Size field must be set to the size of data placed in the buffer as returned from `tkCaptureFn`
- Output values:
  - Score of match is returned by biometric devices in range from 0 to 100. Smart Card genre devices always return 100 in this field if a match is found and 0 if not. Value is required to return only in cBuffer and is “Don’t Care” in iBuffer.

Error codes

TOK_FUNCTION_NOT_SUPPORTED	Can be returned only if called at device level.
TOK _NOT_INITIALIZED	
TOK _WRONG_ARGUMENTS	
TOK _WRONG_DEVICE	
TOK_NO_MATCH_FOUND	
TOK_MATCH_INSUFFICIENT	

Function 0x41

`tkValidateFn` ***(DWORD moduleDescriptor, DWORD functionDescriptor, fpByte cBuffer)***

Table 77: cBuffer for tkValidateFn Function

Size	Name	Comment
Word	Size of data in this buffer in Bytes excluding size and tag fields [1]	Input
Word	Tag [2]	
16 Bytes	User Name space padded [2]	
8 Bytes	Time and Date stamps [2]	
Byte	User Privilege Level [2]	
Byte	Score	Output
Word	Attribute [2]	
Vary	Template Identifier	Input

NOTES:

1. This value excludes size of Tag and Size fields.
2. These fields are used by caller of this function. Function ignores these values and must not alter them.
3. Function descriptor for this function is 0x00043041.
4. Function is optional but if used all parameters are required.

Description of function

This function is used in conjunction with `tkEnrollFn` to perform match of captured and stored templates when the AuD has a "smart" subsystem and internal storage. The function uses the template identifier in cBuffer to identify the template located in the internal storage. The match operation is also performed internally and score of match is returned along with any errors. This function is optional but must be implemented if `tkEnrollFn` is implemented. Information may be also returned by the smart sub-system that can be used to verify the integrity of the MATCH Boolean.

Arguments

Device byte of Module Descriptor must be set to 0 to call function at library level or to device number to call function at device level. Device byte is constructed as described in Section 4.4.4

Buffer fields

- Input values:
  - Size field must be set to the size of data placed in the buffer as returned from `tkEnrollFn`
- Output values:
  - Score of match is returned by biometric devices in range from 0 to 100. Smart Card genre devices always return 100 in this field if a match is found and 0 if not.

Error codes

TOK_FUNCTION_NOT_SUPPORTED	Can be returned only if called at device level.
TOK _NOT_INITIALIZED	
TOK _WRONG_ARGUMENTS	
TOK _WRONG_DEVICE	
TOK_TOKEN_INVALID	
TOK_TEMPLATE_INVALID	
TOK_PIN_REQUIRED	
TOK_PIN_EXPIRED	
TOK_IMAGE_QUALITY_FAILED	
TOK_TEMPLATE_LOW_SEQURITY	
TOK_READ_FAILED	
TOK_NO_MATCH_FOUND	
TOK_MATCH_INSUFFICIENT	

6.7 Function sets

6.7.1 Storage SPL

Table 79 below shows the matrix of the possible sets of functions that can be implemented by Storage SPL vendor depending on type and capabilities of Storage Device as well as accepted security protocol. Matrix reflects functions implemented at library level only. Any of function sets A can be combined with any of sets X – Y, I – III, 1– 2 and P–Q to create whole supported function set.

Table 78: Function Sets

Function	Func#	Set Number									
		A	X	Y	I	II	III	1	2	P	Q
StGetCapabilitiesFn	0x00	X									
StGetInfoFn [1]	0x01	X									
StInitializeFn [1]	0x10	X									
StGetFirstFn	0x20	X									
StGetNextFn	0x21	X									
StGetFirstNameFn	0x22	X									
StGetNextNameFn	0x23	X									
StGetDscrFn	0x24	X									
StAllocateFn	0x30	X									
StFreeFn	0x31	X									
StReadFn	0x50	X									
StWriteFn	0x51	X									
StDeinitializeFn	0x11			X							
StGetConfigFn	0x02					X	X				
StSetConfigFn	0x12						X				
StSetAccessFn	0x40								X		
StSetRightsFn	0x41								X		
StSetLocksFn	0x42										X

NOTES:

1. Some subfunctions are optional at library level.
2. Function group A. This group includes set of mandatory functions .
3. Function groups X–Y. Theses groups include variants of implementation of stDeinitializeFn
   X. Function is not implemented.
   Y. Function inplemented.
4. Function groups I – III. These groups include variants of implementation of BIOS assisted Storage configuration functions.
   I. None of the functions implemented. Either standard BIOS support exists and always available or Storage SPL can enable necessary interface itself
   II stGetConfigFn is implemented. BIOS calls it to get enabling and configuration requirements. stSetConfigFn is not implemented because either there is only one configuration possible or Storage SPL can autodetect enabled devices.
   III. Full set of functions is implemented and BIOS calls stSetConfigFn function to inform Storage SPL about selected configuration.

5. Function groups 1 – 2. These groups include variants of implementation of stSetAccessFn and stSetRightsFn functions.
   1. Both functions are not implemented because “None” security protocol is used
   2. Any other security protocol requires both functions to be implemented.
6. Function groups P – Q include possible implementation of stSetLocksFn function
   P. Storage doesn’t support any locking mechanism. Function is not implemented
   Q. Function is implemented because storage supports some form of locking.

6.7.2 Token SPL

Table 79 below shows the matrix of the possible sets of functions that can be implemented by Token SPL vendor depending on type and capabilities of AuD. Any of sets A – B can be combined with any of sets X – Z, I – III and 1– 3 to create whole supported function set.

Table 79: Function Sets

Function	Func#	Set Number											
		A	B	X	Y	Z	I	II	III	1	2	3	4
TkGetCapabilitiesFn	0x00	X	X										
TkGetInfoFn	0x01	X	X										
TkInitializeFn	0x10	X	X										
TkDeinitializeFn	0x11		X										
TkGetConfigFn	0x02				X	X							
TkSetConfigFn	0x12					X							
TkLoginFn	0x20						X						
TkLoginExtFn	0x21							X					
TkCaptureFn	0x30									X	X	X	
TkRegisterFn	0x31											X	
TkEnrollFn	0x32												X
tkEraseFn	0x33												X
TkMatchFn	0x40										X	X	
TkValidateFn	0x41												X

NOTES:

1. Function groups A – B. These groups include set of mandatory functions 0x00 – 0x11 and differ in implementation of the optional tkDeinitializeFn function.
2. Function groups X – Z. These groups include variants of implementation of BIOS assisted Token configuration functions.
   X. None of the functions implemented. Either standard BIOS exists and always available or Token SPL can enable necessary interface itself
   Y. tkGetConfigFn is implemented. BIOS calls it to get enabling and configuration requirements. tkSetConfigFn is not implemented because either there is only one configuration possible or Token SPL can autodetect enabled devices.
   Z. Full set of functions is implemented and BIOS calls tkSetConfigFn function to inform Token SPL about selected configuration.
3. Function groups I – III. These groups include variants of implementation of Login functions
   I. tkLoginFn is implemented. For example this implementation is necessary for PIN protected Smart Card.
   II. tkLoginExtFn is implemented. The first example of this implementation is for not PIN protected Smart Card. The second example is when Smart Card reader with embedded Key Pad. Bus controller obtains User PIN and unlocks Smart Card. All needed logic is contained in microcode of bus controller. tkLoginExtFn only initiates this process.
   III. None of Login functions is implemented because device is always opened. Example is Fingerprint sensor on LPC bus.
4. Function groups 1 – 4 include possible combinations of capture and match functions
   1. Only tkCaptureFn function is implemented. Example of Token is not PIN protected Smart Card with relevant User information. BIOS needs only capture the User information and then performs match

operation by direct comparison of captured data with content of storage.

2. Function pair tkCaptureFn and tkMatchFn is implemented. Example of Token that needs such support is any biometric AuD. The User Templates captured by such devices don't coincide on 100% and can be matched only by the special matching function.

3. tkRegisterFn, tkCaptureFn and tkMatchFn are implemented. Example is Token that needs such support is biometric AuD when Token Vendor chooses to use different algorithm for capturing of User Template at the time of enrollment then at the time of validation. TkRegisterFn in such case is used for obtaining quality tempalte. It may be interactive, involve multiple samples etc. Obtained template is stored in protected storage. TkCaptureFn thus implements alleviated algorithm which is used for user authentication only. If obtained template doesn't have sufficient quolity, user is simply requested to re-authenticate.

4. Functions tkEnrollFn, tkEraseFn and tkValidateFn are implemented. Example of Token that needs such support is smart AuD with local storage. The EnrolllFn stores User Template locally and returns key which is stored in system Storage. The tkValidateFn captures another User Template and then uses passed key to identify locally stored Template and then performs match operation.

7.0 Reference

7.1 Storage API Interface

Table 80 shows the User Authentication Data Slot assigned to Token ABC with allocated User Templates. This allocation is assumed in the following Sections

Table 80: User Template Layout

Name	Fields	Field Names	Comment
Authentication slot #0 of ABC Token #0	Name	ABC Token	
	Descriptor	00xx80xx	
	Template	Size [1]	Template #0
		Tag	
		Data	
	Template	Size [1]	Template #1
		Tag	
		Data	
		Value 0x00000000 or end of slot [1]	0 Doubleword or end of slot terminate slot scan

NOTES:

1. First template can be found at offset 0 of slot data area. Next template starts at offset Size of Data + 4 etc. Scan terminates when value of Size and Tag fields equals 0 or end of slot is reached.

7.1.1 Token SPL Enumeration

The BIOS will enumerate the Token SPLs by reading and enumerating of all configuration slots using Module Descriptor in the slot name. Configuration slots are assumed to be pre-allocated when Tokens are registered in system. This way the number of registered (supported) Token Stacks can be returned.

7.1.2 User Enumeration

The BIOS will enumerate all enrolled users for every Token by scanning all user authentication data slots registered for a specific Token SPL.

The BIOS will also scan slots using non-zero section size fields in every slot. This way the number of enrolled users for specific Token SPL can be returned.

7.1.3 Token SPL Loading

In case of multiple Token support and a number of enrolled users, the user can be presented with a menu allowing him to select the Token. The selected Token SPL will be loaded by the BIOS using a Module Descriptor of corresponding configuration slot. For alternate media the system must support necessary media access methods.

When needed, the system-wide configuration slots are used to provide additional media information access.

7.1.4 Storing of the User Template

Storing of the user template is part of enrollment for the new user. This process can be originated from the BIOS Setup or from the post OS utility. The user authentication slots are assumed to be pre-allocated when Tokens are registered in system and left blank. The template returned by `tkCaptureFn` or `tkEnrollFn` will be appended to the user authentication slot after the last template. After slot is full the template is written to next slot etc. The capacity of allocated slots limits number of enrolled users.

7.1.5 Erasing the User Template

Erasing the User Template can be originated from the BIOS Setup or from post OS utility.

If the template to be erased is the last one in the slot, BIOS will erase it. If the template is not the last one in slot, the BIOS will overwrite it with the last template and will erase the last template. This method eliminates the need in "garbage collection." If Token SPL supports `tkEraseFn` then it must be called as well. Other methods can also be implemented.

7.1.6 Intel Protected Access Architecture AIC#1 (Anti-replay Integrity Channel)

Intel Security Protocol uses next security objects.

- Pass-phrases are used as Access Control Objects. Use of pass-phrases to ensure data security and integrity is described in Sections 7.1.9 and 7.1.11
- "Nonces" are used as an Protection Challenge Objects. Use of nonces is described in Sections 7.1.8 and 7.1.11.
- "Pchecks" are used as Security Trailers. Use of Pchecks is described in Sections 7.1.10 and 7.1.11

7.1.7 Structure of Buffers

The structure of the header buffer applicable to AIC#1 protocol is shown in Table 81. It is designed to pass all parameters of requested slot to the function. If any data must be written to the slot or returned from the slot they are returned in the Data Buffer – see Table 82.

Table 81: Header Buffer Format for ISP

Structure Size	Structure Name	Field Size	Field Name	Comments
20 bytes	Name Header	16 Bytes	Slot Name [1]	Total 60 bytes Input & output as specified in each function.
		Doubleword	Slot Descriptor [2]	
28 bytes	Security Header	20 Bytes	Pchallenge [3]	
		Doubleword	Access Index [4]	
12 bytes	Buffer Parameter	Doubleword	Handle [5]	
		Doubleword	Offset [6]	
		Doubleword	Size [7]	

Table 82: Data Buffer Format for ISP

Field Size	Field Description	Comments
Vary	Data [9]	Input & output
20 bytes	Pcheck [10]	Input & output

NOTES:

1. Slot Name is free-form string that matches the name of slot owner. This name is not unique – instead it clusters together all slots belonging to specific application. Slot name can be used by application to identify own slots.
2. Slot descriptor is unique doubleword identifier intended to be used by system. Descriptor fields match Module Descriptor format and allow system to classify slots by application and by type of data stored in them.
3. Pchallenge (IPAA Challenge is nonce used to prevent replay of commands – see Figure 5. Nonce is random or pseudo-random number that is used for the duration of only one command/reply sequence. Then nonce is discarded. Rules for use of nonces are specified in Section 7.1.8.
4. Access Index is sequential number of Access Control Object used to compute Security Trailer of input data.The use of Access Indexs is shown in Table 10.
5. Handle is internal identifier used by Storage SPL and is not interpreted by calling application. Internal logic of Handle is incremental by nature. By incrementing of handle Storage SPL can internally enumerate all slots in storage. Rules for use of handles are specified in later sections. All slot related functions return handle that generally must be used as an input parameter for the next call to the same slot.
6. Offset is slot offset of data to operate upon. It is used by data access functions. All enumeration functions don't use and don't change offset. Any value set in this field by caller is preserved by these functions.
7. Size is size of data to oprate upon. It is used by data access functions and is preserved by all other.
8. Data to be read from Storage or written into it.
9. Pcheck is cryptographic hash computed from input and output data with use of shared secrets – pass-phrases.

7.1.8 Use of Protection Challenges and Checks (Pchallenge, Pcheck)

The storage interface employs a security protocol to prevent replay and tampering of data as it traverses between end-points. In its simplest form, a Pchallenge is a nonce that is sent to the "other" end-point as a challenge. A nonce is random number that is only used once during the sequence of calls. The Pchallenge and a shared secret are used to key a SHA-1 hash over the data set to be returned. The result is the Pcheck.

The sequence is as follows:

- The caller makes initialization call to one of the Storage SPL Enumeration Functions – see Section 5.6. This call returns the first nonce, generated by Storage SPL, returned to the caller.
- The caller receives this nonce and saves it as an "Update Nonce" .
- Then the caller prepares the first call to the function that uses nonce – Caller generates "Validation Nonce" and inserts it in Header Buffer.
- Caller saves generated Validation Nonce
- Caller secures data using Update Nonce retrieved and saved from previous initialization call and then passes all buffers to called function.
- When reply is received from Storage SPL the Caller validates it using saved Validation Nonce.
- After validation the Caller retrieves the new Update Nonce from reply and saved for data securing purposes.
- Use of AIC#1 is optional method of Anti-replay protection. If not used the corresponding "Protection Challenge Object Size" field of buffer returned by stGetInfoFn function will contain 0 value – see Table 29

7.1.9 Use of Pass-phrases

The AIC#1 protocol makes the assumption that the best level of protection between two software end-points can't get any better than that provided by pass-phrases known by both end-points (shared secret). This being the case, it is assumed that there are at least 3 pass-phrases known to the system which are described next:

- Pass-phrase 0 (PP 0) is associated with storage itself and is called Administrative Pass-phrase. Generally it is used for Storage-wide administrative tasks
- Each slot additionally has 2 Pass-phrases associated with it - Pass-phrases 1 and 2 (PP 1 and PP 2). Theses Pass-phrases are used for Slot-oriented operations.

When access to slot is queried the *Index of Pass-phrase* used in the query must be passed to Storage SPL in the Header Buffer.

Every Pass-phrase has associated bit-mask of Access Rights or permissions. This bit-mask has bits controlling all which actions the associated access pass-phrase is allowed to do within a particular slot. When data access or administrative function is called the corresponding *permission* bit is consulted and function proceeds only if it is in the "YES" state.

A pass-phrase used to query access to slot is not passed "in clear". Instead it is used along with content of all buffers to compute Pcheck – cryptographic hash. Recipient uses Access Index to discover used Pass-phrase. It then can validate data integrity and caller identity by recalculation of Pcheck and comparing of obtained and received values. Access is granted only if Pcheck values match. See more about it in the Section 7.1.10

7.1.10 Use of Pchecks

Storage interface uses PCheck values to provide temper resistance and to control access to storage data. Pchecks are cryptographic hashes of interface data. Interface data are concatenation of next elements.

- Pass–phrase;
- Update Pcheck
- Content of all command or reply buffers –Header buffer and Data Buffer.

Table 83 shows the order of how data are concatenated for computation of Pchecks and place where Pchecks are positioned.

Table 83: Computation of Pchecks

Size	Component Name	Field Name	Comments
20 bytes	Pass-phrase		These data are hashed in specified order
20 bytes	Pchallenge (Update nonce)		
60 bytes	Header Buffer		
Vary	Data Buffer	Data	
20 bytes		Computed Pcheck	New Pcheck is inserted at the data end

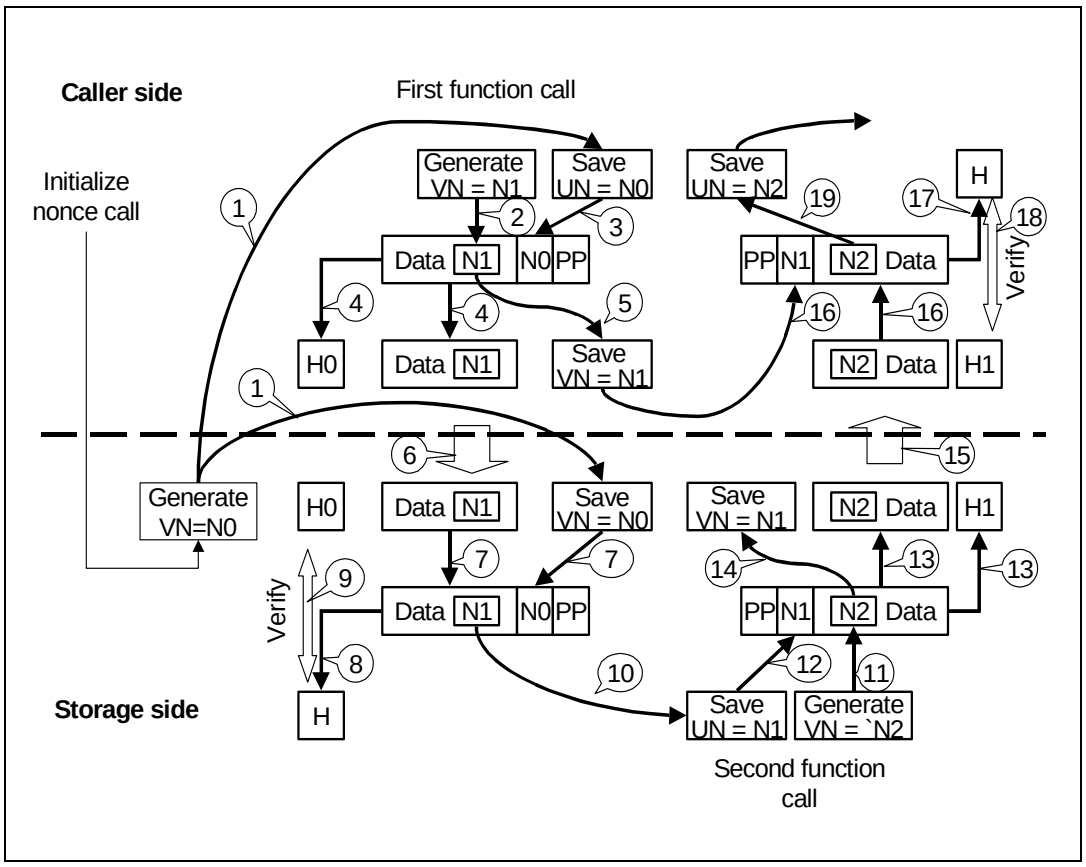
Pchecks are used as follows.

- All elements to be hashed are concatenated in specified above order;
- The standard hash value is computed and placed at the end of Data Buffer.
- Command buffers (Header and Data) are sent to Recipient;
- Recipient validates data integrity by computing of the same Pcheck value using Validation Nonce stored during previous function call and Pass–phrase stored permanently in slot header and referred to by Access Index. If data were tampered or incorrect Pass–phrase used by Caller to hash data this validation will fail.

7.1.11 Communication Flow

Communication Flow with use of Pchallenges and Pchecks is illustrated on Figure 5.

Figure 5: Use of Pchallenges and Pchecks



NOTES:

1. N0, N1 – nonce # 0, # 1 etc.
2. H0, H1 – hash value # 0, #1
3. VN – Validation Nonce. Nonce used by recipient for data validation.
4. UN – Update Nonce. Nonce used by caller for hashing of data to be sent.
5. PP – Pass Phrase. Shared secret phrase known to caller and recipient. Used for authorrization of data access.

Sequence of events on Figure 5.

1. Nonce N0 is generated. Function used for setting up of initial nonce values is called. Generated value of N0 is saved as update nonce UN on caller side and as verify nonce VN on storage side.
2. Nonce N1 is generated and inserted into data buffer.
3. Data are concatenated with saved in step 1 nonce N0 and pass phrase PP to create common data buffer DD = Data + N0 + PP.
4. Hash H0 is generated out of buffer DD
5. N1 is saved as validation nonce VN for subsequent verification of reply value.
6. Data access function is called and both data DD and Hash H0 are sent to storage side.

7. On storage side received data DD are concatenated with saved in step 1 validation nonce N0 and PP.
8. Hash H is calculated.
9. H is compared with H0 for data validation.
10. Nonce N1 received in data buffer is saved as update nonce UN.
11. Nonce N2 is generated (repeat of step 2) and is inserted in reply data.
12. Data are concatenated with saved in step 10 nonce N1 and pass phrase PP to create common data buffer DD = Data + N1 + PP (repeat of step 3).
13. Hash H1 is generated out of buffer DD (repeat of step 4)
14. N2 is saved as verification nonce VN for subsequent verification of next call value (repeat of step 5).
15. Data access function is completed and both data DD and hash H1 are returned to caller side (repeat of step 6).
16. On caller side received data DD are concatenated with saved in step 5 validation nonce N1 and PP (repeat of step 7).
17. Hash H is calculated (repeat of step 8).
18. H is compared with H1 for data validation (repeat of step 9).
19. Nonce N2 received in data buffer is saved as update nonce UN (repeat of step 10).

7.2 "None" Protocol

"None" Protocol doesn't use any means of data protection between end-points. It relies on hardware Platform and Chipset capabilities. This doesn't preclude the SPL from implementing its own protection protocol between the SPL and the hardware.

Next tables show Header and Data Buffers used by functions when None Protocol is in use.

Table 84: Header Buffer Format for "None" Protocol

Structure Size	Structure Name	Field Size	Field Name	Comments
20 bytes	Name Header	16 Bytes	Slot Name [1]	Total 60 bytes
		Doubleword	Slot Descriptor [2]	Input & output as specified in each function.
Vary	Security Header [6]	Vary	Vary	
12 bytes	Buffer Parameter	Doubleword	Handle [3]	
		Doubleword	Offset [4]	
		Doubleword	Size [5]	

Table 85: Data Buffer Format for "None" Protocol

	Field Description	Comments
Vary	Data [6]	Input & output

NOTES:

1. Slot Name is free-form string that matches the name of slot owner. This name is not unique – instead it clusters together all slots belonging to specific application. Slot name can be used by application to identify own slots.
2. Slot descriptor is unique doubleword identifier intended to be used by system. Descriptor fields match Module Descriptor format and allow system to classify slots by application and by type of data stored in them.
3. Handle is internal identifier used by Storage SPL and is not interpreted by calling application. Internal logic of Handle is incremental by nature. By incrementing of handle Storage SPL can internally enumerate all slots in storage. Rules for use of handles are specified in later sections. All slot related functions return handle that generally must be used as an input parameter for the next call to the same slot.
4. Offset is slot offset of data to operate upon. It is used by data access functions. All enumeration functions don't use and don't change offset. Any value set in this field by caller is preserved by these functions.
5. Size is size of data to oprate upon. It is used by data access functions and is preserved by all other.
6. Data to be read from Storage or written into it
7. Security Header is optional when "None" Protocol is used with Storage Locking as a mean of data protection.

When "None" Protocol is used, the corresponding Size fields of all Security Objects in the buffer returned by stGetInfoFn function, must contain 0 value (except for the Access Control Object field)– see Table 29.

7.3 Run-time Module handling

Because all modules of the Security System need to be loaded and linked at run time, a facility has to be defined that allows run time identification of necessary modules, locating them, placing in low or high memory, decompression, relocation, and establishing of run-time links. This goal can be achieved by prefixing modules with headers. Headers will expose necessary information for loader, decompressor and relocator and will export entry points for run-time execution.

Because all modules are assumed to be compressed (except a decompression module itself) the first level header must be not compressed for loader analysis.

The phases of file loading are:

1. Find the module with specified attributes on specified media and read first level header. Determine source and destination addresses, compressed and expanded sizes, and size of relocation header.
2. Decompress module with relocation header into destination address of memory with proper alignment.
3. Relocate the module.
4. Register and fix-up entry points for run time execution.

7.3.1 Common Module Organization

Table 86 specifies the file layout that satisfies the previously mentioned requirements.

Table 86: Module Layout

Header Name	Usage	Comment
Prefix	Used for loading [1]	Not Compressed
Relocation Header	Used for relocating [2]	Compressed
Export Structure	Used for linking [3]	
Module Body	Module code [3]	

NOTES:

1. The prefix will be stripped out by the loader.
2. The relocation header will be stripped out by the relocator.
3. At run-time only the Export Structure and the file body aligned at a 1-K boundary will be present in memory.

7.3.2 Loading of Modules

7.3.2.1 Prefix

The following Prefix provides the necessary information for the loader, the decompressor, and the relocator.

Table 87: Prefix Layout

Name	Marker	Length	Prefix Checksum	Module Descriptor	Compressed Size	Expanded Size	Header Size	Expanded File Checksum	Attribute
Size	Double word	Byte	Byte	Double word	Double word	Double word	Double word	Double word	Word
Value	$$$$	28	Vary	Vary	Vary	Vary	Vary	Vary	Vary

7.3.2.2 Module Attribute

To load the module, the loader needs to identify the module specific requirements for placement – low or high memory placement, bitness, compression type, and the presence of the relocation header. The following File Attribute satisfies the above requirement:

Table 88: Module Attribute

Bit #	Value	Name	Comment
0	1	Module is 32-bit (if 0 – 16-bit)	
1	1	Protected mode Module (if 0 – real mode)	
2	1	Module must be placed in low memory	
3		Reserved	
4	1	Module requires fix-up	
5 – 7	0	Relocation header in EXE format	
	1	Relocation header in PE format	
	2	Relocation header in REX format	
	3 – 8	Reserved	
8		Reserved	
9		Reserved	
10	1	Module is compressed	
11–14	0	Implode compression method	
	1	LZSS compression method	
	2	LZARI compression method	
	3 –15	Reserved	
15		Reserved	

7.3.2.3 Other Prefix Fields

- Compressed and expanded sizes of the module are used to decompress it and to place it into memory.
- The size of the relocation header is used to align the Export Structure onto a 1-K boundary. Although this information as well as header type information can be obtained from analysis of the header itself, adding of this field to prefix will simplify loader code and logic.
- Expanded file check-sum is used to verify the module integrity after decompression.
- The prefix length and check-sum are used to verify the integrity of the prefix itself. It is assumed that check-sum field will contain byte-wide 2's complement of sum of all other prefix bytes. To verify the prefix integrity a caller will have to extract value from length byte and perform byte-wide summing of all prefix bytes on specified length. This sum will yield zero value.

7.3.2.4 Building a Module

In order to produce a module with the above organization, a third step of modifying the module binary image is necessary besides the two standard development steps of compiling and linking. This step will check-sum module, compress it, attach a prefix, insert the required information into the prefix fields, and check-sum the prefix.

It is assumed that the decompressed module will be supplied in binary form by Token SPL Vendor and its modification be performed by engineer responsible for integrating of Token SPL into BIOS.

7.3.3 Relocation of Modules

Generally each module in protected mode can be executed in two different ways conforming to two different memory models –"flat" and "segmented".

For a flat memory model, after module is loaded into memory, it must be fixed-up in such a way that all internal offsets are relative to a '0' memory offset. To perform such fix-ups, the Loader must find the relocation table in the module header and for each item in this table add the base address of memory where the module is placed to the value of the item. After such fix-up, the Relocation Header can be discarded. The module entry point can be access with a NEAR CALL using only its 32-bit offset.

Thus, relocation of a module for the flat memory model always requires fix-ups and presence of Relocation Header.

For a segmented memory model, the module is linked in such a way that all internal offsets are relative to the beginning of the module itself. No run-time fix-ups are needed for such a module. To make a call, the caller will construct a descriptor with base address equal to base address of module in memory, load the code segment register with the selector of this descriptor and make a far call to the module entry point.

Therefore relocation of the module in the segmented memory model doesn't require fix-ups and Relocation Header. Additionally, to conserve media space, the Header can be stripped-out at design time.

In this document it is assumed that all modules are executed in segmented memory model and that module organization is simplified as in the following table.

Table 89: Simplified Module Layout

Header Name	Usage	Comment
Prefix	Used for loading [1]	Not Compressed
Export Structure	Used for linking [2]	Compressed
Module Body	Module code [2]	

NOTES:

1. The prefix will be stripped out by the loader
2. At run-time only the Export Structure and the file body aligned at a 1-K boundary will be present in memory.

7.4 Specific Requirements

7.4.1 S3 Wake from Sleep State Support

Strong system security requires user re-authentication upon system wake from the S3 through the S5 sleep states. A system wake from the S4 and the S5 sleep states is similar to a cold boot in the use of system memory. Upon these wakes, the content of system memory is either not retained from the previous full-on state or it is restored from HDD. In any case for some time the BIOS possesses control of the whole system memory and can allocate it for Security Operations.

A system wake from the S3 sleep state is different because memory content is retained from the pre-sleep state and therefore the BIOS will not gain any memory unless it is reserved by the previous cold boot.

In order to reserve such memory, two memory functions of int 15 – e801 and e820 must be modified. Function e801 must exclude reserved memory from return value. Function e820 must return reserved memory region as memory type 2.

In order to be OS compliant, this reserved memory has to be relative to the Top of Installed Memory. Therefore, actual memory addresses for BIOS use can be determined only at run time. This imposes a requirement for all security system modules to be re-locatable.

7.4.2 Pre-boot Specific Requirements

This section summarizes the pre-boot environment's specific requirements to any executed code.

- Typically, the pre-boot environment does not support dynamic memory allocation. The pre-boot application should use only statically allocated data.
- The pre-boot environment does not support paging, processes, threads, and at this point there is no intent to emulate these or other Windows specific features. The application must not use any of the library functions including C run-time library that may be using the mentioned features. Generally, the application should not be linked to C standard libraries, or the results of such linking have to be carefully analyzed.
- The application must ensure that Start-up and Exit code (added to the application by Linker) is not executed.
- The application should not use any of Int 21 services. Only BIOS supported standard services can be used.

There are additional requirements for applications running in 32-bit protected mode.

- The protected mode pre-boot application should assume that there is a special core code running on the system and use only the defaults and services provided on the application's behalf.
- The protected mode pre-boot application should assume that all its code and all its statically allocated data are **NEAR** and therefore cannot contain any FAR CALLS and FAR JUMPS
- If far pointers to data are passed to the application, it is the core's responsibility to verify that the segments pointed to are present in memory and that passed selectors are valid and are read/write capable.
- The application should not use any software interrupts either provided by the OS or the BIOS except those supported by core.

The following are additional requirements that can essentially simplify development of pre-boot applications.

- The application should use one common contiguous CODE segment and one common initialized DATA segment.
- Both CODE and DATA segments should have the same Base Address equal to 0 relatively to the beginning of application file CODE segment.
- The application should not allocate or switch stacks. Common full 64-K stack will be maintained by the core on behalf of all running applications. This stack will have Big bit Set when 32-bit application is executed. The stack will be placed in low memory to make it available equally for real-mode and protected-mode applications.

7.5 Manufacturing Support

7.5.1 Possible implementations of Storage

Next are examples of Storage implementation

- One or two banks of the Firmware Hub, which is part of Intel chip-set, can be used as protected storage. Firmware Hub banks are individually access controlled. Each of the banks can be set as read-protected, write-protected or both and locked down in this state. A bank which is locked down can be unlocked only after hardware reset. All these features provide a data protection mechanism which can be implemented through BIOS.
- Smart Card with PIN protected access to elementary files that serve as storage. This PIN must be individual for every platform and must be stored locally.
- Special Storage chip can be designed to serve as Storage.

7.5.2 Manufacturer or IT First Boot

Before the first boot the Storage must be present in system and be in the initial state without users enrolled. Note that initial state must be different from the totally erased state. When the system is first booted, the BIOS will encounter no user enrolled in the Storage and will boot to OS. At this time the OS present utility can be used to brand Storage and Storage SPL. The branding process must leave Storage in the initial state while changing the Storage SPL and Storage data in such a way that both become unique amidst all of the manufactured systems. This is to prevent replay attacks and should support a mode that does not yield any personally identifiable information.

7.5.3 Second Boot

When the system is booted for the second time, the BIOS will encounter no users enrolled and will boot to OS. Depending on the implementation, either an OS present or BIOS Setup utility can be used to enroll the first user.

7.5.4 Third Boot

When system is booted for the third time, it will recognize enrolled user(s) and will prompt for user to authenticate.

Appendix A: BIOS Memory Layout

The following list identifies the typical allocation of memory at pre-boot time.

- Segment 00000h – Interrupt Vector Table and BIOS stack
- Segment 01000h – Used for decompression of BIOS during shadowing
- Segment 02000h – BIOS memory manager structures
- Segment 03000h – PnP Structures and BIOS Setup strings
- Segment 04000h – BIOS Setup
- Segment 05000h – 09000h Available
- Segment 0A000h – Video buffer
- Segment 0B000h – Video buffer and Upper memory block
- Segment 0C000h – Video Bios and option ROMs
- Segment 0D000h – Upper memory Blocks and Mapped System Management Memory in SMM
- Segment 0E000h – BIOS ROM and Mapped System Management Memory in SMM
- Segment 0F000h – BIOS ROM

The Security System can allocate typically 5 * 64K = 320K of conventional memory for Security Services. This value can be different for specific BIOS's but in most cases this amount needs to be considered as an available maximum.

Expected Memory Requirements

- One of the Token Stacks – Finger Print Sensor Stack used for validation of Security principles consumed as much as 450K in decompressed form for Stack code and statically allocated data. Although this is an extreme size, the Stack is expected to require more than 64K.
- The expected size of protected storage driver is 32K.
- A BIOS Applet has an expected size of up to 16K.
- The same size is expected for decompression module and policy manager.
- A full 64-K independent stack will be necessary.

For a system that supports three Tokens, these estimates yield 64 * 3 + 32 + 16 * 3 + 64 = 336K. Future expansions may require even more memory and therefore the design cannot be based on the traditional use on conventional memory.

Expected Flash Part Size

All sizes below are for compressed components.

- AT BIOS and Applet – 128K
- PXE/BIS – 70K
- Video BIOS – 64K
- Protected Storage Stack – 32K
- Token Stacks – 3 * 64K = 192K
- Core , Policy Manager, Decompression Module – 20K

The Total is about 506 KBytes

Appendix B: Example Zero-based Segments "C" style

Appendix B contains a fragment of a C 6.0 MAKE file that produces 0 based CODE and DATA segments.

```
#----------------------------------------------------------------------------
#Common directories
ABINDIR = ..\TOOL\MASM\BIN
CBINDIR = ..\TOOL\MSVC60\BIN
COMMDIR = ..\COMMON
PROJDIR = ..\$(PROJ)

#----------------------------------------------------------------------------
#Build tools
AS = $(ABINDIR)\ML
CC = $(CBINDIR)\CL
LN = $(CBINDIR)\LINK
MAKE = $(CBINDIR)\NMAKER

#----------------------------------------------------------------------------
#Flags
LFLAGS0=/NOLOGO /SUBSYSTEM:CONSOLE /ENTRY:Dispatch /BASE:0
/NODEFAULTLIB
LFLAGS1 = /MAP:$(PROJ).MAP /MAPINFO:EXPORTS /MAPINFO:FIXUPS
/OUT:$(PROJ).EXE
CFLAGS0 = /c /nologo /Gd /Gs /Gy /G4 /Zp1 /W1 /Od /Zi
CFLAGS1 = /I $(INCDIR)
AFLAGS = /c /Cu /Fl /Sa /I$(COMMDIR) /D"BUILD=_BIOS" /coff

LFLAGS=$(LFLAGS0) $(LFLAGS1)
CFLAGS = $(CFLAGS0) $(CFLAGS1)

#----------------------------------------------------------------------------
#Map files
MAPFILE = $(PROJ).MAP

#----------------------------------------------------------------------------
#Libs
LIBS =
```

```
#-----------------------------------------------------------------------------
#Object files
OBJFILES =VFPSMAIN.OBJ\
          VFPSC.OBJ

#-----------------------------------------------------------------------------
#Include files
INCFILES = VFPSMAIN.INC\
           $(COMMDIR)\COMMON.INC
HFILES=COMDEFS.H

#-----------------------------------------------------------------------------
#This is main target for building of this directory
all:      all1 $(PROJ).EXE
all1:
          del $(PROJ).LOG
          echo.> $(PROJ).LOG
          echo >NUL @<<$(PROJ).FLG
$(CFLAGS)
<<NOKEEP

$(PROJ).EXE: $(OBJFILES)
          echo >NUL @<<$(PROJ).CRF
$(LFLAGS)
$(OBJFILES)
<<NOKEEP
          $(LN) @$(PROJ).CRF > $(PROJ).ERL
          if ERRORLEVEL == 0 DEL $(PROJ).ERL

#-----------------------------------------------------------------------------
#Specific dependencies
$(PROJ).EXE:      $(OBJFILES)
VFPSMAIN.OBJ: VFPSMAIN.ASM $(INCFILES)
          $(AS) $(AFLAGS) $*.ASM > $*.ERR
          if ERRORLEVEL == 0 DEL $*.ERR

VFPSC.OBJ:      VFPSC.C $(HFILES)
          $(CC) /FlVFPSC.LST @$(PROJ).FLG $*.C >> $(PROJ).LOG
```

Appendix C: Example Export Structure "MASM" Style

Appendix C contains fragment of MASM code that shows how to initialize Export Structure at run time.

```
;;@MakeArray macro returns quoted string for initialization of
;array structure fields
;
@MakeArray MACRO string
        LOCAL str
        LOCAL chr
        LOCAL iter
        LOCAL i
        iter = @SizeStr(<string>)
        i = 1
        str TEXTEQU <>
        chr TEXTEQU <>
        WHILE iter

        chr SUBSTR <&string>,i,1

        str CATSTR str,<’>,chr,<’,>
        iter = iter - 1
        i = i + 1
        ENDM
        i= @SizeStr(%str)
        str SUBSTR str,1,i-1
        exitm str
ENDM
;————————————————————————————————
;File Name
;
vfpsName TEXTEQU <vfps>
vfpsNameQ CATSTR <’>, vfpsName, <’>
vfpsNameArr TEXTEQU @MakeArray(<%vfpsName>)

;————————————————————————————————-
; This must start from beginning of file
;
displacement =    OFFSET Dispatch-OFFSET fileHeaderStart
fileHeaderStart LABEL BYTE

fileHeader exportStructure          {{vfpsNameArr}, \
                                    , \
                                    CSum, \
                                    displacement, \
                                    0, \
                                    0}
```

```
CSum = 0
% FORC par, <vfpsName>
        CSum = CSum + '&par'
ENDM
CSum = CSum + LOW(SIZEOF(exportStructure))
CSum = CSum + HIGH(SIZEOF(exportStructure))
CSum = CSum + LOW(displacement)
CSum = CSum + HIGH(displacement)
CSum = (NOT (CSum)) + 1

;------------------------------------------------------------------------------
;       Dispatch
;
;       Entry:
;       Exit:
;       Modified:
;       Processing:
;       Errors:

Dispatch PROC FAR PUBLIC TokNum:DWORD, FuncNum: DWORD, extBuf:FPBYTE32
;
;Etc.
;
```

Appendix D: Error Code Listing

Appendix D lists the main error codes.

Error codes.

//Base values	Comments
ERROR_BASE = 0x80000000	
TOK_ERROR_BASE = ERROR_BASE + 0x10000	
STO_ERROR_BASE = ERROR_BASE + 0x20000	
//Common errors of Storage SPL	
STO_FUNCTION_NOT_SUPPORTED = STO_ERROR_BASE + 0x0	Requested function is not supporrted at library or device level
STO_NOT_INITIALIZED = STO_ERROR_BASE + 0x10	Function requires storage to be initialized
//Interface errors	
STO_WRONG_ARGUMENTS = STO_ERROR_BASE + 0x100	Returned for any determined distortion of arguments that is not superseed by specific error code
STO_SMALL_BUFFER = STO_ERROR_BASE + 0x110	Return data don't fit into allocated buffer.
STO_WRONG_DEVICE = STO_ERROR_BASE + 0x120	Device byte of Module descriptor is set to wrong value
STO_WRONG_NAME = STO_ERROR_BASE + 0x140	Storage slot name differs from header buffer value for specified handle
STO_WRONG_DESC = STO_ERROR_BASE + 0x150	Storage slot descriptor differs from header buffer value for specified handle
STO_WRONG_HANDLE = STO_ERROR_BASE + 0x160	Storage doesn't have slot that correspond to specified handle
//Information functions	
//Initialization functions	
STO_INITIALIZATION_FAILED = STO_ERROR_BASE + 0x300	Storage initialization function failed at library or device level. This error code is returned if not superseeded by specific error codes.
//Enumeration functions	
STO_SLOT_NOT_FOUND = STO_ERROR_BASE + 0x400	Requested slot not found
//Allocation functions	
STO_NO_FREE_SLOTS = STO_ERROR_BASE + 0x500	There is no room to allocate requiested slot
STO_DESC_NOT_UNIQUE = STO_ERROR_BASE + 0x510	
//Administration functions	
STO_INSUFFICIENT_RIGHTS = STO_ERROR_BASE + 0x600	Access rights are not sufficient for requested operation.
STO_LOCK_FAILED = STO_ERROR_BASE + 0x610	Media failed locking.
STO_LOCK_NOT_SUPPORTED = STO_ERROR_BASE + 0x620	Media doesn't support requested lock type
//Data access functions	
STO_ACCESS_DENIED = STO_ERROR_BASE + 0x700	Access to slot data is not granted for caller
STO_READ_FAILED = STO_ERROR_BASE + 0x710	Media read error – signifies HW mulfunction
STO_WRITE_FAILED = STO_ERROR_BASE + 0x720	Media write error – signifies HW mulfunction. Error returned if not superseeded by specific code
STO_ERASE_FAILED = STO_ERROR_BASE + 0x730	Media erase error – signifies HW mulfunction

//Custom errors	
STO_CUSTOM_ERROR_BASE = STO_ERROR_BASE + 0x800	Custom errors
//Unknown error	
STO_UNKNOWN_ERROR = STO_ERROR_BASE + 0xF00	Not specified Storage SPL error
//Common errors of Token SPL	
TOK_FUNCTION_NOT_SUPPORTED = TOK_ERROR_BASE + 0x0	Requested function is not supporrted at library or device level
TOK_NOT_INITIALIZED = TOK_ERROR_BASE + 0x10	Function requires token to be initialized
//Interface errors	
TOK_WRONG_ARGUMENTS = TOK_ERROR_BASE + 0x100	Returned for any determined distortion of arguments that is not superseed by specific error code
TOK_SMALL_BUFFER = TOK_ERROR_BASE + 0x110	Return data don't fit into allocated buffer.
TOK_WRONG_DEVICE = STO_ERROR_BASE + 0x120	Device byte of Module descriptor is set to wrong value
//Information functions	
//Initialization functions	
TOK_INITIALIZATION_FAILED = TOK_ERROR_BASE + 0x300	Storage initialization function failed at library or device level. This error code is returned if not superseeded by specific error codes.
TOK_TOKEN_NOT_FOUND = TOK_ERROR_BASE + 0x350	Authentication device cannot be found on interface bus.
TOK_MORE_THAN_ONE_TOKEN= TOK_ERROR_BASE + 0x360	More than one valid authentication device is attached.
//Login Functions	
TOK_TOKEN_INVALID = TOK_ERROR_BASE + 0x400	Device is found but authentication information is not present
TOK_TEMPLATE_INVALID = TOK_ERROR_BASE + 0x410	Authentication information is corrupted.
TOK_LOGIN_FAILED = TOK_ERROR_BASE + 0x420	Authentication information is PIN protected and PIN verification failed.
//Capture Functions	
TOK_DEVICE_READ_FAILED = TOK_ERROR_BASE + 0x500	Device read operation failed. Signifies interface or HW mulfunction. Returned if not superseeded by specific error codes.
TOK_WRITE_FAILED= TOK_ERROR_BASE + 0x510	For Enroll function template write operation failed
TOK_TEMPLATE_NOT_FOUND = TOK_ERROR_BASE + 0x520	For erase function template to be erased not found
TOK_PIN_REQUIRED = TOK_ERROR_BASE + 0x530	Device is PIN protected
TOK_PIN_EXPIRED = TOK_ERROR_BASE + 0x540	PIN lockout is actrive and must be reset by administrator
TOK_IMAGE_QUALITY_FAILED = TOK_ERROR_BASE + 0x580	Quolity of raw image is too low to extract template
TOK_TEMPLATE_LOW_SEQURITY = TOK_ERROR_BASE + 0x590	Template contains too few minutiae to be used for authentication
//Verification Functions	
TOK_NO_MATCH_FOUND = TOK_ERROR_BASE + 0x600	Template don't match – score is lower than match criteria
TOK_MATCH_INSUFFICIENT = TOK_ERROR_BASE + 0x610	Template match but score is lower than threshold value
//Custom errors	
TOK_CUSTOM_ERROR_BASE = TOK_ERROR_BASE + 0x800	Custom errors
//Unknown error	
TOK_UNKNOWN_ERROR = TOK_ERROR_BASE + 0xF00	Not specified Token SPL error

Appendix E: Main Tag Listing

Appendix E lists the main tags.

Tags.

//Base values	Comments
TAG_BASE_ STANDARD = 0	Base of assigned tags
TAG_BASE_ CUSTOM = 0x8000	Cutom tags available for SP and BIOS developers
//Common tags	
TAG_NO_TAG = 0	No tag or tag not assigned
TAG_TOKEN_FRIENDLY_NAME = 0x1	Name of Token SP for display purposes.
TAG_CONFIGURATION_DATA = 0x2	SP configuration data.
TAG_CUSTOM_ERROR_STRING = 0x3	SP custom error message.
TAG_TEMPLATE_SIZE=0X4	SP size of template
TAG_SYSTEM_POLICY=0X100	System policy section
TAG_USER_TEMPLATE = 0x1000	User Template structure

Appendix F: SHA-1 based ACO protection

This appendix describes the SHA-1 hash based one-time pad method for changing the access control object in a protected storage slot. This method provides protection against snooping at the interface level and is most effective if the originating endpoint is operating within a security perimeter with a know level of protection and assurance.

Note1: this mechanism only works with ACOs that are 160bits or less.

Note2: This mechanism must be accompanied by a good integrity mechanism, such as rolling nonce keyed HMAC described in this specification, to prevent a form of Denial-of-Service where an attacker changes the ACO to some random value.

Setting a New ACO

When setting a new slot or administrator access control object (ACO), the new ACO can be obscured using the known current ACO to generate a one-time pad as shown below.

1. Create a 64-byte string, *ipad*, each byte of which contains the value 0x36.
2. Modify the string by performing a bitwise XOR of the most significant bits of *ipad* with the current access control object, *CurACO* The resulting string is called *istring*.
3. Perform a SHA-1 hash of *istring*. The result is the 20-byte SHA-1 hash digest, *h1*.
4. Create a 64-byte string, *opad*, each byte of which contains the value 0x5C.
5. Modify the string by performing a bitwise XOR of the most signifcant bits of *opad* with the current access control object *CurACO*. ase *CurPassPhrase*. The resulting string is called *ostring*.
6. Perform a SHA-1 hash of the concatenation of *ostring* followed by *h1*. The result is a 20-byte SHA-1 hash digest ACOPV. The one-timepad.
7. The new ACO is then obscured by performing a bit-wise XOR with this one-time pad (ACOPV).

That is:

$$ACOPV = SHA1((CurACO \oplus opad) \,||\, SHA1(CurACO \oplus ipad))$$

$$ObscuredACO = ACOPV \oplus NewACO$$

Where,

⊕	The bit-wise exclusive-or operator.
||	The data concatenation operator.
SHA1(x)	The SHA-1 hash function of data string x
CurACO	The current ACO pointed to by ACO Index
ipad	A string containing the byte 0x36 repeated 64 times.
opad	A string containing the byte 0x5C repeated 64 times.
ACOPV	Access Control Object Protection Value
NewACO	The new ACO pointed to by ACO Index
ObscuredACO	The obscured ACO that will be sent to the protected storage sub-system to replace the old one.